CMU Common Lisp User's Manual

Robert A. MacLachlan, *Editor*

February 1991

CMU-CS-91-108 2

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This is a revised version of Technical Report CMU-CS-87-156.

Abstract

CMU Common Lisp is an implementation of Common Lisp that currently runs under Mach, a Berkeley Unix 4.3 binary compatible operating system. CMU Common Lisp is currently supported on MIPS-processor DECstations, Sparc-based workstations from Sun and the IBM RT PC, and other ports are planned. The largest single part of this document describes the Python compiler and the programming styles and techniques that the compiler encourages. The rest of the document describes extensions and the implementation dependent choices made in developing this implementation of Common Lisp. We have added several extensions, including the proposed error system, a source level debugger, an interface to Mach system calls, a foreign function call interface, support for interprocess communication and remote procedure call, and other features that provide a good environment for developing Lisp code.

This research was sponsored by the Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title *Research on Parallel Computing* issued by DARPA/CMO under Contract MDA972-90-C-0035 ARPA Order No. 7330.

Keywords: Programming environments, Common Lisp, system interface, debuggers, compilers, efficiency, Python

Table of Contents

Chapter 1

Introduction

CMU Common Lisp is a public-domain implementation of Common Lisp developed in the Computer Science Department of Carnegie Mellon University. CMU Common Lisp is currently supported on MIPS-processor DECstations, Sparc-based workstations from Sun and the IBM RT PC, and other ports are planned. Currently, it runs only under CMU's Mach operating system. This document describes the implementation based on the Python compiler. Previous versions of CMU Common Lisp ran on the IBM RT PC and (when known as Spice Lisp) on the Perq workstation.

This manual contains only implementation-specific information about CMU Common Lisp. Users will also need a separate manual describing the COMMON LISP standard. COMMON LISP was initially defined in *Common Lisp: The Language*, by Guy L. Steele Jr. COMMON LISP is now undergoing standardization by the X3J13 committee of ANSI. The X3J13 spec is not yet completed, but a number of clarifications and modification have been approved. We intend that CMU Common Lisp will eventually adhere to the X3J13 spec, and we have already implemented many of the changes approved by X3J13.

Until the X3J13 standard is completed, the second edition of *Common Lisp: The Language* is probably the best available manual for the language and for our implmentation of it. This book has no official role in the standardization process, but it does include many of the changes adopted since the first edition was completed.

In addition to the language itself, this document describes a number of useful library modules that run in CMU Common Lisp. Hemlock, an Emacs-like text editor, is included as an integral part of the CMU Common Lisp environment. Two documents describe Hemlock: the *Hemlock User's Manual*, and the *Hemlock Command Implementor's Manual.*

CMU Common Lisp is currently undergoing intensive tuning and development. For the next year or so, new releases will be appearing frequently. This document will be updated for each major release. Users of CMU Common Lisp at CMU should watch the Mach, Unix-Announce, Unix-Forum, and Clisp bulletin boards for release announcements, pointers to updated documentation files, and other information of interest.

1.1. Obtaining and Running CMU Common Lisp under Mach

To run CMU Common Lisp you must have a supported workstation with at least 8 megabytes of memory (the more the better, especially if you run X.) The Hemlock editor supports the workstation's high-resolution display under the X window manager, as well as standard terminals such as the Concept-100 or H-19.

At CMU, there is a misc collection named `cs.misc.cmucl` which should be updated on your machine regularly by normal `sup` mechanisms. You can run "`lisp`" out of AFS, but you must have an unusually large AFS cache to do so (20 megabytes or more.) For those outside of CMU, there are several files including "`lisp`",

"lisp.core", "spelldict.bin", etc. that need to be installed. "lisp" is a C program that loads "lisp.core" into memory. "lisp" should be put in any bin directory that is normally in your search path. "lisp" currently expects to find "lisp.core" in the directory "/usr/misc/.cmucl/lib/". If Hemlock is run under the X window system, it needs several files that it expects to find in this this directory. The X inspector also looks for files in "/usr/misc/.cmucl/lib/", and expects to find a help file in the directory "/usr/misc/.cmucl/doc/".

At CMU, you should put either "/usr/misc/bin" (if you want all the misc executable files) or "/usr/misc/.cmucl/bin" (if you want just Common Lisp) in your PATH searchlist. Typing "lisp" will start up Lisp with the default core image ("/usr/misc/.lisp/lib/lisp.core") after several seconds.

Currently Lisp accepts the following switches:

-core	requires an argument that should be the name of a core file. Rather than using the default core file ("/usr/misc/.lisp/lib/lisp.core"), the specified core file is loaded.
-edit	specifies to enter Hemlock. A file to edit may be specified by placing the name of the file between the program name (usually **lisp**) and the first switch.
-eval	accepts one argument which should be a Lisp form to evaluate during the start up sequence. The value of the form will not be printed unless it is wrapped in a form that does output.
-hinit	accepts an argument that should be the name of the hemlock init file to load the first time the function **ed** is invoked. The default is to load "**hemlock-init.***object-type*", or if that does not exist, "**hemlock-init.lisp**" from the user's home directory. If the file is not in the user's home directory, the full path must be specified.
-init	accepts an argument that should be the name of an init file to load during the normal start up sequence. The default is to load "**init.***object-type*" or, if that does not exist, "**init.lisp**" from the user's home directory. If the file is not in the user's home directory, the full path must be specified.
-noinit	accepts no arguments and specifies that an init file should not be loaded during the normal start up sequence. Also, this switch suppresses the loading of a hemlock init file when Hemlock is started up with the **-edit** switch.
-load	accepts an argument which should be the name of a file to load into Lisp before entering Lisp's read-eval-print loop.
-slave	specifies that Lisp should start up as a slave Lisp and try to connect to an editor Lisp. The name of the editor to connect to must be specified — to find the editor's name, use the Hemlock "**Accept Slave Connections**" command. The name for the editor Lisp is of the form "*machine-name*:*socket*", where *machine-name* is the internet host name for the machine and *socket* is the decimal number of the socket to connect to.

For more details on the use of the **-edit** and **-slave** switches, see the *Hemlock User's Manual.*

Arguments to the above switches can be specified in one of two ways: *switch=value* or *switch<space>value*. For example, to start up the saved core file mylisp.core use either of the following two commands:

```
lisp -core=mylisp.core
lisp -core mylisp.core
```

Chapter 2

Implementation Dependent Design Choices

Several design choices in Common Lisp are left to the individual implementation. This chapter contains a partial list of these topics and the choices that are implemented in CMU Common Lisp. As in *Common Lisp: The Language*, all symbols and package names are printed in lower case, as a user is likely to type them. Internally, they are normally stored upper case only.

2.1. Integers

The **fixnum** type is equivalent to **(signed-byte 30)**. Integers outside this range are represented as a **bignum** or a word integer (see section 7.9.6.) Almost all integers that appear in programs can be represented as a **fixnum**, so integer number consing is rare.

2.2. Floats

CMU Common Lisp supports two floating point formats: **single-float** and **double-float**. These are implemented with IEEE single and double float arithmetic, respectively. **short-float** is a synonym for **single-float**, and **long-float** is a synonym for **double-float**. The initial value of ***read-default-float-format*** is **single-float**.

Both **single-float** and **double-float** are represented with a pointer descriptor, so float operations can cause number consing. Number consing is greatly reduced if programs are written to allow the use of non-descriptor representations (see section 7.9.)

2.2.1. IEEE Special Values

CMU Common Lisp supports the IEEE infinity and NaN special values. These non-numeric values will only be generated when trapping is disabled for some floating point exception (see section 2.2.4), so users of the default configuration need not concern themselves with special values.

extensions:short-float-positive-infinity [*Constant*]
extensions:short-float-negative-infinity [*Constant*]
extensions:single-float-positive-infinity [*Constant*]
extensions:single-float-negative-infinity [*Constant*]
extensions:double-float-positive-infinity [*Constant*]
extensions:double-float-negative-infinity [*Constant*]
extensions:long-float-positive-infinity [*Constant*]
extensions:long-float-negative-infinity [*Constant*]

The values of these constants are the IEEE positive and negative infinity objects for each float format.

extensions:float-infinity-p *x* [*Function*]
This function returns true if *x* is an IEEE float infinity (of either sign.) *x* must be a float.

extensions:float-nan-p *x* [*Function*]
extensions:float-trapping-nan-p *x* [*Function*]
float-nan-p returns true if *x* is an IEEE NaN (Not A Number) object. **float-trapping-nan-p** returns true only if *x* is a trapping NaN. With either function, *x* must be a float.

2.2.2. Negative Zero

The IEEE float format provides for distinct positive and negative zeros. To test the sign on zero (or any other float), use the COMMON LISP **float-sign** function. Negative zero prints as `-0.0f0` or `-0.0d0`.

2.2.3. Denormalized Floats

CMU Common Lisp supports IEEE denormalized floats. Denormalized floats provide a mechanism for gradual underflow. The COMMON LISP **float-precision** function returns the actual precision of a denormalized float, which will be less than **float-digits**. Note that in order to generate (or even print) denormalized floats, trapping must be disabled for the underflow exception (see section 2.2.4.) The COMMON LISP **least-positive-***format***float** constants are denormalized.

extensions:float-normalized-p *x* [*Function*]
This function returns true if *x* is a denormalized float. *x* must be a float.

2.2.4. Floating Point Exceptions

The IEEE floating point standard defines several exceptions that occur when the result of a floating point operation is unclear or undesirable. Exceptions can be ignored, in which case some default action is taken, such as returning a special value. When trapping is enabled for an exception, a error is signalled whenever that exception occurs. These are the possible floating point exceptions:

:underflow This exception occurs when the result of an operation is too small to be represented as a normalized float in its format. If trapping is enabled, the **floating-point-underflow** condition is signalled. Otherwise, the operation results in a denormalized float or zero.

:overflow This exception occurs when the result of an operation is too large to be represented as a float in its format. If trapping is enabled, the **floating-point-overflow** exception is signalled. Otherwise, the operation results in the appropriate infinity.

:inexact This exception occurs when the result of a floating point operation is not exact, i.e. the result was rounded. If trapping is enabled, the **extensions:floating-point-inexact** condition is signalled. Otherwise, the rounded result is returned.

:invalid This exception occurs when the result of an operation is ill-defined, such as `(/ 0.0 0.0)`. If trapping is enabled, the **extensions:floating-point-invalid** condition is signalled. Otherwise, a quiet NaN is returned.

:divide-by-zero
This exception occurs when a float is divided by zero. If trapping is enabled, the **divide-by-zero** condition is signalled. Otherwise, the appropriate infinity is returned.

2.2.5. Floating Point Rounding Mode

IEEE floating point specifies four possible rounding modes:

`:nearest` In this mode, the inexact results are rounded to the nearer of the two possible result values. If the neither possibility is nearer, then the even alternative is chosen. This form of rounding is also called "round to even", and is the form of rounding specified for the COMMON LISP `round` function.

`:positive-infinity`
This mode rounds inexact results to the possible value closer to positive infinity. This is analogous to the COMMON LISP `ceiling` function.

`:negative-infinity`
This mode rounds inexact results to the possible value closer to negative infinity. This is analogous to the COMMON LISP `floor` function.

`:zero` This mode rounds inexact results to the possible value closer to zero. This is analogous to the COMMON LISP `truncate` function.

Warning:

Although the rounding mode can be changed with `set-floating-point-modes`, use of any value other than the default (`:nearest`) can cause unusual behavior, since it will affect rounding done by Common Lisp system code as well as rounding in user code. In particular, the unary `round` function will stop doing round-to-nearest on floats, and instead do the selected form of rounding.

2.2.6. Accessing the Floating Point Modes

These functions can be used to modify or read the floating point modes:

`extensions:set-floating-point-modes &key :traps :rounding-mode :fast-mode :accrued-exceptions :current-exceptions` [*Function*]

`get-floating-point-modes` [*Function*]

The keyword arguments to `set-floating-point-modes` set various modes controlling how floating point arithmetic is done:

`:traps` A list of the exception conditions that should cause traps. Possible exceptions are `:underflow`, `:overflow`, `:inexact`, `:invalid` and `:divide-by-zero`. Initially all traps except `:inexact` are enabled. See section 2.2.4.

`:rounding-mode`
The rounding mode to use when the result is not exact. Possible values are `:nearest`, `:positive-infinity`, `:negative-infinity` and `:zero`. Initially, the rounding mode is `:nearest`. See the warning in section 2.2.5 about use of other rounding modes.

`:current-exceptions :accrued-exceptions>`
Lists of exception keywords used to set the exception flags. The *current-exceptions* are the exceptions for the previous operation, so setting it is not very useful. The *accrued-exceptions* are a cumulative record of the exceptions that occurred since the last time these flags were cleared. Specifying `()` will clear any accrued exceptions.

`:fast-mode` Set the hardware's "fast mode" flag, if any. When set, IEEE conformance or debuggability may be impaired. Some machines may not have this feature, in which case the value is always `nil`. No currently supported machines have a fast mode.

If a keyword argument is not supplied, then the associated state is not changed.

get-floating-point-modes returns a list representing the state of the floating point modes. The list is in the same format as the keyword arguments to **set-floating-point-modes**, so **apply** could be used with **set-floating-point-modes** to restore the modes in effect at the time of the call to **get-floating-point-modes**.

2.3. Characters

CMU Common Lisp implements characters according to *Common Lisp: the Language II*. The main difference from the first version is that character bits and font have been eliminated, and the names of the types have been changed. **base-character** is the new equivalent of the old **string-char**. In this implementation, all characters are base characters (there are no extended characters.) Character codes range between 0 and 255, using the ASCII encoding.

2.4. Array Initialization

If no **:initial-value** is specified, arrays are initialized to zero.

2.5. Packages

When CMU Common Lisp is first started up, the default package is the **user** package. The **user** package uses the **lisp**, **extensions**, **conditions**, **debug**, and **clos** packages. The symbols exported from these five packages can be referenced without package qualifiers.

Currently, the following packages are defined (abbreviations for the packages are in parenthesis after the full name):

clos (pcl) The **clos** package contains the code that implements the Common Lisp Object System (CLOS) specification and exports the symbols as defined in the CLOS specification. The nickname **pcl** has been retained for compatibility with earlier versions.

c The **c** package contains the Common Lisp compiler. User-level compiler extensions are exported from the **extensions** package.

conditions The **conditions** package contains the new error system as proposed for Common Lisp and exports several symbols necessary for the new error system.

debug The **debug** package contains the stack crawling debugger and the low level functions on which it is built. It exports symbols the user may want to use when debugging a program.

dired The **dired** package contains support functions for Hemlock's directory editing mode.

extensions (ext) The **extensions** packages exports local extensions to Common Lisp that are documented in this manual. Examples include the **save-lisp** function and the interface to foreign (C) functions.

hemlock (ed) The **hemlock** package contains all the code to implement Hemlock commands. The **hemlock** package currently exports no symbols.

hemlock-internals (hi) The **hemlock-internals** package contains code that implements low level primitives and exports those symbols used to write Hemlock commands.

inspect The **inspect** package contains the inspector.

iterate The **iterate** package contains code used by CLOS and exports a few symbols needed by CLOS.

keyword	The **keyword** package contains keywords (e.g., :start). All symbols in the **keyword** package are exported and evaluate to themselves (i.e., the value of the symbol is the symbol itself).
lisp	The **lisp** package exports all the symbols defined by *Common Lisp: the Language* and only those symbols. Strictly portable Lisp code will depend only on the symbols exported from the **lisp** package.
mach	The **mach** package contains code to interface to the Mach operating system. All the standard unix system calls (the names are unix-<system call name>) and the Mach specific calls (e.g., vm_allocate, port_allocate, etc.) are exported from this package.
spell	The **spell** package contains a spelling checker and corrector that is used by Hemlock. It exports several symbols that allow a user to manipulate the spelling dictionary and to check the spelling of words.
system (sys)	The **system** package contains functions and information necessary for the system. This package is used by the **lisp** package and exports several symbols that are necessary to interface to system code. For example, the symbols used by the alien facility are exported from this package.
user	The **user** package is the default package and is where a user's code and data is placed unless otherwise specified. The **user** package exports no symbols.
walker	The **walker** package contains code used by CLOS and exports a few symbols needed by CLOS.
xlib	The **xlib** package contains the Common Lisp X interface (CLX) to the X11 protocol. This is mostly Lisp code with a couple of functions that are defined in C to connect to the server.
xp	The **xp** package contains a version of Richard C. Waters's pretty printer written in Common Lisp.

The **lisp**, **user** and **keyword** packages are required by COMMON LISP.

2.6. The Editor

The **ed** function invokes the Hemlock editor which is described in *Hemlock User's Manual* and *Hemlock Command Implementor's Manual*. Most users at CMU prefer to use Hemlock's slave Common Lisp mechanism which provides an interactive buffer for the **read-eval-print** loop and editor commands for evaluating and compiling text from a buffer into the slave Common Lisp. Since the editor runs in the Common Lisp, using slaves keeps users from trashing their editor by developing in the same Common Lisp with Hemlock.

2.7. Time Functions

time *form* [*Macro*]

This macro evaluates *form*, prints some timing and memory allocation information to ***trace-output***, and returns any values that *form* returns. The timing information includes real time, user run time, and system run time. The consing information is useful for relative comparisons, but for accurate memory allocation reporting, you should compile the call to **time** in a dummy function and call the function.

internal-time-units-per-second [*Constant*]

The value of internal-time-units-per-second is 100.

See section 7.12 for more information.

2.8. Garbage Collection

CMU Common Lisp uses a stop-and-copy garbage collector that compacts the items in dynamic space every time it runs. Most users cause the system to garbage collect (GC) frequently, long before space is exhausted. With eight or twelve megabytes of memory, causing GC's more frequently on less garbage allows the system to GC without much (if any) paging.

The following functions invoke the garbage collector or control whether automatic garbage collection is in effect:

extensions:gc [*Function*]
This function runs the garbage collector. If **ext:*gc-verbose*** is non-**nil**, then it invokes **ext:*gc-notify-before*** before GC'ing and **ext:*gc-notify-after*** afterwards.

extensions:gc-off [*Function*]
This function inhibits automatic garbage collection. After calling it, the system will not GC unless you call **ext:gc** or **ext:gc-on**.

extensions:gc-on [*Function*]
This function reinstates automatic garbage collection. If the system would have GC'ed while automatic GC was inhibited, then this will call **ext:gc**.

The following variables control the behavior of the garbage collector:

extensions:*bytes-consed-between-gcs* [*Variable*]
CMU Common Lisp automatically GC's whenever the amount of memory allocated to dynamic objects exceeds the value of an internal variable. After each GC, the system sets this internal variable to the amount of dynamic space in use at that point plus the value of the variable **ext:*bytes-consed-between-gcs***. The default value is 2000000.

extensions:*gc-verbose* [*Variable*]
This variable controls whether **ext:gc** invokes the functions in **ext:*gc-notify-before*** and **ext:*gc-notify-after***. If ****gc-verbose**** is **nil**, **ext:gc** foregoes printing any messages. The default value is **t**.

extensions:*gc-notify-before* [*Variable*]
This variable's value is a function that should notify the user that the system is about to GC. It takes one argument, the amount of dynamic space in use before the GC measured in bytes. The default value of this variable is a function that prints a message similar to the following:

[GC threshold exceeded with 2,107,124 bytes in use. Commencing GC.]

extensions:*gc-notify-after* [*Variable*]
This variable's value is a function that should notify the user when a GC finishes. The function must take three arguments, the amount of dynamic spaced retained by the GC, the amount of dynamic space freed, and the new threshold which is the minimum amount of space in use before the next GC will occur. All values are byte quantities. The default value of this variable is a function that prints a message similar to the following:

[GC completed with 25,680 bytes retained and 2,096,808 bytes freed.]
[GC will next occur when at least 2,025,680 bytes are in use.]

Note that a garbage collection will not happen at exactly the new threshold printed by the default `ext:*gc-notify-after*` function. The system periodically checks whether this threshold has been exceeded, and only then does a garbage collection.

`extensions:*gc-inhibit-hook*` [*Variable*]

This variable's value is either a function of one argument or `nil`. When the system has triggered an automatic GC, if this variable is a function, then the system calls the function with the amount of dynamic space currently in use (measured in bytes). If the function returns `nil`, then the GC occurs; otherwise, the system inhibits automatic GC as if you had called `ext:gc-off`. The writer of this hook is responsible for knowing when automatic GC has been turned off and for calling or providing a way to call `ext:gc-on`. The default value of this variable is `nil`.

`extensions:*before-gc-hooks*` [*Variable*]
`extensions:*after-gc-hooks*` [*Variable*]

These variables' values are lists of functions to call before or after any GC occurs. The system provides these purely for side-effect, and the functions take no arguments.

2.9. Describe

In addition to the basic function described below, there are a number of switches and other things that can be used to control `describe`'s behavior.

`describe` *object* `&optional` *stream* [*Function*]

The `describe` function prints useful information about *object* on *stream*, which defaults to `*standard-output*`. For any object, `describe` will print out the type. Then it prints other information based on the type of `object`. The types which are presently handled are:

`hash-table`	`describe` prints the number of entries currently in the hash table and the number of buckets currently allocated.
`function`	`describe` prints a list of the function's name (if any) and its formal parameters. If the name has function documentation, then it will be printed. If the function is compiled, then the file where it is defined will be printed as well.
`fixnum`	`describe` prints whether the integer is prime or not.
`symbol`	The symbol's value, properties, and documentation are printed. If the symbol has a function definition, then the function is described.

If there is anything interesting to be said about some component of the object, describe will invoke itself recursively to describe that object. The level of recursion is indicated by indenting output.

`extensions:*describe-level*` [*Variable*]

The maximum level of recursive description allowed. Initially two.

`extensions:*describe-indentation*` [*Variable*]

The number of spaces to indent for each level of recursive description, initially three.

`extensions:*describe-print-level*` [*Variable*]
`extensions:*describe-print-length*` [*Variable*]

The values of `*print-level*` and `*print-length*` during description. Initially two and five.

2.10. Load

An extension has been made to **load** to allow the user to control what happens when the object file is older than the corresponding source file.

extensions:*load-if-source-newer* [*Variable*]

The legal values for ***load-if-source-newer*** and their meanings are:

:load-object	The object file is loaded even though the source file is newer. This is the default.
:load-source	The source file is loaded instead of the older object file.
:compile	The source file is compiled and then the new object file is loaded.
:query	The user is asked a yes or no question to determine whether the source or object file is loaded.

If ***load-if-source-newer*** contains any other value, an error is signalled.

2.11. The Inspector

An inspector that runs under the X window system, version 11, or on a terminal is available in CMU Common Lisp.

inspect &optional *object* [*Function*]

Inspect calls the inspector on the optional argument *object*. If *object* is unsupplied, **inspect** immediately returns **nil**. Otherwise, the behavior of inspect depends on whether Lisp is running under X.

If X is available, **inspect** creates an X window and displays *object* in the window. While **inspect** is running and the cursor is in the inspector's X window, mouse clicks and keyboard input have the following meaning:

Left	When the left mouse button is clicked over a component object, that object will be inspected in the current inspector window.
Middle	When the middle mouse button is clicked over a component object, **inspect** is exited returning the component as the result. All the new inspector windows are deleted.
Shift Middle	When the shift key is depressed and the middle mouse button is clicked over a component object, **inspect** exits and returns the component as the result. All the inspector windows are left displayed on the screen.
Right	When the right mouse button is clicked over a component object, that object will be inspected in a new inspector window.
d, D	When either d or D is typed, the current window is deleted. If there are no more windows, then **inspect** exits and returns the original *object*.
h, H, ?	When any of h, H, or ? are typed while in an inspector window, a new window with help information is displayed.
m, M	When either m or M is typed, a component object may be modified. The cursor changes to an arrow with an M beside it. Clicking any mouse button while the mouse is over a component will select that component as the destination for modification. If m was typed, the source object is also selected by the mouse which is indicated by an S beside the arrow in the cursor. If M was typed, the source object will be prompted for on the *query-io* stream. The source object replaces the destination object. While choosing the destination or source with the mouse, the operation can be aborted by type q or Q.

q, Q	When either q or Q is typed, `inspect` exits and returns the original *object*. All new inspector windows are deleted.
p, P	When either p or P is typed, `inspect` exits and returns the original *object*. All the inspector windows are left on the screen.
r, R	When either r or R is typed, the current inspector display is recomputed. This is necessary to maintain a consistent display for an object that may have changed since the display was originally computed.
u, U	When either u or U is typed, the object of which the current object is a component is displayed. This is the inverse operation to clicking the left mouse button over a component object. If the window is currently displaying the top level object, nothing changes.

When the cursor is over a component object, the object is highlighted with a surrounding box.

If X is unavailable, a terminal inspector is invoked. This inspector prints information about and object and a numbered list of the components of the object. The following commands are available:

<n>	where <n> means a number corresponding to one of the components of the object. The inspector changes its focus to be this component. The inspector displays the components of the this new object.
r	recomputes the information for the current object.
d	redisplays the information for the current object.
u	moves up one level of the objects inspected. As you descend into the components of an object, a stack of all the objects previously seen is kept. This command pops you up one level of this stack.
q, e	quits the inspector returning the currently inspected object.
h, ?	displays some help text.

When `inspect` is eventually exited, it returns a Lisp object.

Chapter 3

Miscellaneous Extensions to Common Lisp

The developers of CMU Common Lisp have added several extensions to make the system a better development environment. This chapter describes these functions, macros, and variables that add to the basic Common Lisp.

3.1. Unix Interrupts

CMU Common Lisp allows access to all the Unix signals that can be generated under Mach. It should be noted that if this capability is abused, it is possible to completely destroy the running Lisp. The following macros and functions allow access to the Unix interrupt system. The signal names as specified in section 2 of the *Unix Programmer's Manual* are exported from the Mach package.

system:with-enabled-interrupts *specs* **&rest** *body* [*Macro*]
This macro should be called with a list of signal specifications, *specs*. Each element of *specs* should be a list of two elements: the first should be the Unix signal for which a handler should be established, the second should be a function to be called when the signal is received One or more signal handlers can be established in this way. **with-enabled-interrupts** establishes the correct signal handlers and then executes the forms in *body*. The forms are executed in an unwind-protect so that the state of the signal handlers will be restored to what it was before the **with-enabled-interrupts** was entered. A signal handler function specified as NIL will set the Unix signal handler to the default which is normally either to ignore the signal or to cause a core dump depending on the particular signal.

system:without-interrupts &rest *body* [*Macro*]
It is sometimes necessary to execute a piece a code that can not be interrupted. This macro the forms in *body* with interrupts disabled. Note that the Unix interrupts are not actually disabled, rather they are queued until after *body* has finished executing.

system:with-interrupts &rest *body* [*Macro*]
When executing an interrupt handler, the system disables interrupts, as if the handler was wrapped in in a **without-interrupts**. The macro **with-interrupts** can be used to enable interrupts while the forms in *body* are evaluated. This is useful if *body* is going to enter a break loop or do some long computation that might need to be interrupted.

system:without-hemlock &rest *body* [*Macro*]
For some interrupts, such as SIGTSTP (suspend the Lisp process and return to the Unix shell) it is necessary to leave Hemlock and then return to it. This macro executes the forms in *body* after exiting Hemlock. When *body* has been executed, control is returned to Hemlock.

system:enable-interrupt *signal function* [*Function*]

This function establishes *function* as the handler for *signal*. Unless you want to establish a global signal handler, you should use the macro **with-enabled-interrupts** to temporarily establish a signal handler. **enable-interrupt** returns the old function associated with the signal.

system:ignore-interrupt *signal* [*Function*]

Ignore-interrupt sets the Unix signal mechanism to ignore *signal* which means that the Lisp process will never see the signal. Ignore-interrupt returns the old function associated with the signal or **nil** if none is currently defined.

system:default-interrupt *signal* [*Function*]

Default-interrupt can be used to tell the Unix signal mechanism to perform the default action for *signal*. For details on what the default action for a signal is, see section 2 of the *Unix Programmer's Manual*. In general, it is likely to ignore the signal or to cause a core dump.

3.1.1. Default Interrupt Handlers for Lisp

CMU Common Lisp has several interrupt handlers defined when it starts up, as follows:

SIGINT — causes Lisp to enter a break loop. This puts you into the debugger which allows you to look at the current state of the computation. If you proceed from the break loop, the computation will proceed from where it was interrupted.

SIGQUIT — causes Lisp to do a throw to the top-level. This causes the current computation to be aborted, and control returned to the top-level read-eval-print loop.

SIGTSTP — causes Lisp to suspend execution and return to the Unix shell. If control is returned to Lisp, the computation will proceed from where it was interrupted.

SIGILL, SIGBUS, SIGSEGV, and SIGFPE — cause Lisp to signal an error.

The SIGINT, SIGQUIT, and SIGTSTP signals can be generated from the keyboard. The characters used to generate these interrupts are the same as in the shell. Generally, these are control-C for SIGINT, control-\ for SIGQUIT, and control-Z for SIGTSTP. Depending on what commands are in your .login or .cshrc files, the characters used to generate these interrupts may be different. When in the Lisp read-eval-print loop that you get by just running Lisp, these interrupts can be generated by typing the appropriate character. To generate one of these interrupts from the keyboard while running Hemlock depends on how Hemlock is run, as follows:

Under X — When running under the X window manager, SIGINT, SIGQUIT, and SIGTSTP are generated by typing the appropriate control character in the top-level Lisp window.

Terminal — When accessing Lisp from a normal terminal (either by telnet or terminal emulation mode under X), control-\ can be used to generate the SIGINT signal. The other interrupts can not be signalled directly while in Hemlock, but once in the debugger, they can be signalled by typing the appropriate character. Note that the "**Pause Hemlock**" command can be used to pause the Hemlock process.

When a signal is generated, there may be some delay before it is processed since Lisp cannot be interrupted safely in an arbitrary place. The computation will continue until a safe point is reached and then the interrupt will be processed.

Unix signals that correspond to program errors cause the Lisp error system to obtain control. Under normal circumstances this should not happen, but if it does and you have important work, you should immediately try to save it.

3.1.2. Examples of Signal Handlers

The following code is the signal handler used by the Lisp system for the SIGINT signal.

```
(defun ih-sigint (signal code scp)
  (declare (ignore signal code scp))
  (without-hemlock
   (with-interrupts
    (break "Software Interrupt" t))))
```

The `without-hemlock` form is used to make sure that Hemlock is exited before a break loop is entered. The `with-interrupts` form is used to enable interrupts because the user may want to generate an interrupt while in the break loop. Finally, break is called to enter a break loop, so the user can look at the current state of the computation. If the user proceeds from the break loop, the computation will be restarted from where it was interrupted.

The following function is the Lisp signal handler for the SIGTSTP signal which suspends a process and returns to the Unix shell.

```
(defun ih-sigtstp (signal code scp)
  (declare (ignore signal code scp))
  (without-hemlock
   (mach:unix-kill (mach:unix-getpid) mach:sigstop)))
```

Lisp uses this interrupt handler to catch the SIGTSTP signal because it is necessary to get out of Hemlock in a clean way before returning to the shell.

To set up these interrupt handlers, the following is recommended:

```
(with-enabled-interrupts ((mach:SIGINT #'ih-sigint)
                          (mach:SIGTSTP #'ih-sigtstp))
  <user code to execute with the above signal handlers enabled.>
)
```

3.2. Saving a Core Image

A mechanism has been provided to save a running Lisp core image and to later restore it. This is convenient if you don't want to load several files into a Lisp when you first start it up.

`extensions:save-lisp` *file* **`&key :purify :root-structures :init-function :load-init-file :print-herald :process-command-line`** [*Function*]

The **`save-lisp`** function saves the state of the currently running Lisp core image in *file*. The keyword arguments have the following meaning:

:purify — If non-NIL (the default), the core image is purified before it is saved. This means moving accessible Lisp objects from dynamic space into read-only and static space. This reduces the amount of work the garbage collector must do when the resulting core image is being run. Also, if more than one Lisp is running on the same machine, this maximizes the amount of memory that can be shared between the two processes. Objects in read-only and static space can never be reclaimed, even if all pointers to them are dropped.

:root-structures — This should be a list of the main entry points for the resulting core image. The purification process tries to localize symbols, functions, etc., in the core image so that paging performance is improved. The default value is NIL which means that Lisp objects will still be localized but probably not as optimally as they could be. This argument has no meaning if *:purify* is NIL.

:init-function — This is a function which is called when the saved core is resumed. The default

function simply aborts to the top-level read-eval-print loop. If the function returns, it will be the value of **save-lisp**.

:load-init-file If non-NIL, then load an init file; either the one specified on the command line or "**init**.*fasl-type*", or, if "**init**.*fasl-type*" does not exist, "**init.lisp**" from the user's home directory. If the init file is found, it is loaded into the resumed core file before the read-eval-print loop is entered.

:print-herald If non-NIL, then print out the standard Lisp herald when starting.

:process-command-line
If non-NIL, processes the command line switches and performs the appropriate actions.

To resume a saved file, type:

```
lisp -core file
```

3.3. Search Lists

Search lists make it possible to refer to files using abbreviated names. The general form of a search list definition is:

```
(setf (ext:search-list name) '(directory1 directory2 ...))
```

name specifies the search list, and must be a string (case insensitive) terminated by a colon (:). The $directory_i$ are strings that specify Unix directories (case sensitive). For example, it is possible to define the search list "**code:**" as follows:

```
(setf (ext:search-list "code:") '("/usr/lisp/code/"))
```

It is now possible to use "**code:**" as an abbreviation for the directory "**/usr/lisp/code/**" in all file operations. For example, you can now specify "**code:eval.lisp** to refer to the file "**/usr/lisp/code/eval.lisp**".

To obtain the value of a search-list name, use the function search-list as follows:

```
(ext:search-list name)
```

Where *name* is the name of a search list as described above. If *name* is not defined as a search-list, **nil** is returned. For example, calling **ext:search-list** on "**code:**" as follows:

```
(ext:search-list "code:")
```

returns the list **("/usr/lisp/code/")**.

3.4. Running Programs from Lisp

It is possible to run programs from Lisp by using the following function.

extensions:run-program *program args* **&key :env :wait :pty :input :if-input-does-not-exist :output ...** [*Function*]

Run-program runs *program* in a child process. *Program* should be a pathname or string naming the program. *Args* should be a list of strings which this passes to *program* as normal Unix parameters. For no arguments, specify *args* as **nil**.

The value returned is either a process structure or **nil**. The process interface follows the description of **run-program**.

When you are done using a process, call **process-close** to reclaim system resources. You only need

to do this when you supply **:stream** for one of *:input*, *:output*, or *:error*, or you supply **:pty** non-**nil**. You can call **process-close** regardless of whether you must to reclaim resources without penalty if you feel safer.

run-program accepts the following keyword arguments:

:env This is an a-list mapping keywords and simple-strings. The default is **ext:*environment-list***. If *:env* is specified, **run-program** uses the value given and does not combine the environment passed to Lisp with the one specified.

:wait If non-**nil** (the default), wait until the child process terminates. If **nil**, continue running Lisp while the child process runs.

:pty This should be one of **t**, **nil**, or a stream. If specified non-**nil**, the subprocess executes under a Unix *PTY*. If specified as a stream, the system collects all output to this pty and writes it to this stream. If specified as **t**, the **process-pty** slot contains a stream from which you can read the program's output and to which you can write input for the program. The default is **nil**.

:input This specifies how the program gets its input. If specified as a string, it is the name of a file that contains input for the child process. **run-program** opens the file as standard input. If specified as **nil** (the default), then standard input is the file "**/dev/null**". If specified as **t**, the program uses the current standard input. This may cause some confusion if *:wait* is **nil** since two processes may use the terminal at the same time. If specified as **:stream**, then the **process-input** slot contains an output stream. Anything written to this stream goes to the program as input. *:Input* may also be an input stream that already contains all the input for the process. In this case **run-program** reads all the input from this stream before returning, so this cannot be used to interact with the process.

:if-input-does-not-exist
This specifies what to do if the input file does not exist. The following values are valid: **nil** (the default) causes **run-program** to return **nil** without doing anything; *:create* creates the named file; and *:error* signals an error.

:output This specifies what happens with the program's output. If specified as a pathname, it is the name of a file that contains output the program writes to its standard output. If specified as **nil** (the default), all output goes to "**/dev/null**". If specified as **t**, the program writes to the Lisp process's standard output. This may cause confusion if *:wait* is **nil** since two processes may write to the terminal at the same time. If specified as **:stream**, then the **process-output** slot contains an input stream from which you can read the program's output.

:if-output-exists This specifies what to do if the output file already exists. The following values are valid: **nil** causes **run-program** to return **nil** without doing anything; *:error* (the default) signals an error; *:supersede* overwrites the current file; and *:append* appends all output to the file.

:error This is similar to *:output*, except the file becomes the program's standard error. Additionally, *:error* can be *:output* in which case the program's error output is routed to the same place specified for *:output*. If specified as **::stream**, the **process-error** contains a stream similar to the **process-output** slot when specifying the *:output* argument.

:if-error-exists This specifies what to do if the error output file already exists. It accepts the same values as *:if-output-exists*.

:status-hook This specifies a function to call whenever the process changes status. This is especially useful when specifying *:wait* as **nil**. The function takes the process as a required argument.

:before-execve This specifies a function to run in the child process before it becomes the program to run. This is useful for actions such as authenticating the child process without modifying the parent Lisp process.

Except for sharing file descriptors as explained in keyword argument descriptions, **run-program** closes all file descriptors in the child process before running the program.

If **run-program** fails to fork the child process, it returns **nil**.

The following functions interface the process returned by **run-program**.

extensions:process-p *thing* [*Function*]
This function returns **t** if *thing* is a process. Otherwise it returns **nil**

extensions:process-pid *process* [*Function*]
This function returns the process ID, an integer, for the *process*.

extensions:process-status *process* [*Function*]
This function returns the current status of *process*, which is one of **:running**, **:stopped**, **:exited**, or **:signaled**.

extensions:process-exit-code *process* [*Function*]
This function returns either the exit code for *process*, if it is **:exited**, or the termination signal *process* if it is **:signaled**. The result is undefined for processes that are still alive.

extensions:process-core-dumped *process* [*Function*]
This function returns **t** if someone used a Unix signal to terminate the *process* and caused it to dump a Unix core image.

extensions:process-pty *process* [*Function*]
This function returns either the two-way stream connected to *process*'s Unix *PTY* connection or **nil** if there is none.

extensions:process-input *process* [*Function*]
This function returns either the output stream connected to *process*'s input or **nil** if there is none.

extensions:process-output *process* [*Function*]
This function returns either the input stream connected to *process*'s output or **nil** if there is none.

extensions:process-error *process* [*Function*]
This function returns either the input stream connected to *process*'s error output or **nil** if there is none.

extensions:process-status-hook *process* [*Function*]
This function returns the current function to call whenever *process*'s status changes. This function takes the *process* as a required argument. **process-status-hook** is **setf**'able.

extensions:process-plist *process* [*Function*]
This function returns annotations supplied by users, and it is **setf**'able. This is available solely for users to associate information with *process* without having to build a-lists or hash tables of process structures.

extensions:process-wait *process* **&optional** *check-for-stopped* [*Function*]

This function waits for *process* to finish. If *check-for-stopped* is non-**nil**, this also returns when *process* stops.

extensions:process-kill *process signal* **&optional** *whom* [*Function*]

This function sends the Unix *signal* to *process*. *Signal* should be the number of the signal or a keyword with the Unix name (for example, **:sigsegv**). *Whom* should be one of the following:

:pid This is the default, and it indicates sending the signal to *process* only.

:process-group
This indicates sending the signal to *process*'s group.

:pty-process-group
This indicates sending the signal to the process group currently in the foreground on the Unix *PTY* connected to *process*. This last option is useful if the running program is a shell, and you wish to signal the program running under the shell, not the shell itself. If **process-pty** of *process* is **nil**, using this option is an error.

extensions:process-alive-p *process* [*Function*]

This function returns **t** if *process*'s status is either **:running** or **:stopped**.

extensions:process-close *process* [*Function*]

This function closes all the streams associated with *process*. When you are done using a process, call this to reclaim system resources.

3.5. Time Parsing and Formatting

Functions are provided to allow parsing strings containing time information and printing time in various formats are available.

extensions:parse-time *time-string* **&key :error-on-mismatch :default-seconds :default-minutes :default-hours :default-day ...** [*Function*]

parse-time accepts a string containing a time (e.g., "**Jan 12, 1952**") and returns the universal time if it is successful. If it is unsuccessful and the keyword argument *:error-on-mismatch* is non-**nil**, it signals an error. Otherwise it returns **nil**. The other keyword arguments have the following meaning:

:default-seconds specifies the default value for the seconds value if one is not provided by *time-string*. The default value is 0.

:default-minutes specifies the default value for the minutes value if one is not provided by *time-string*. The default value is 0.

:default-hours specifies the default value for the hours value if one is not provided by *time-string*. The default value is 0.

:default-day specifies the default value for the day value if one is not provided by *time-string*. The default value is the current day.

:default-month specifies the default value for the month value if one is not provided by *time-string*. The default value is the current month.

:default-year specifies the default value for the year value if one is not provided by *time-string*. The default value is the current year.

:default-zone specifies the default value for the time zone value if one is not provided by *time-string*. The default value is the current time zone.

:default-weekday specifies the default value for the day of the week if one is not provided by *time-string*. The default value is the current day of the week.

Any of the above keywords can be given the value *:current* which means to use the current value as determined by a call to the operating system.

extensions:format-universal-time *dest universal-time* **&key :timezone :style :date-first :print-seconds ...** [*Function*]

extensions:format-decoded-time *dest seconds minutes hours day month year* **&key ...** [*Function*]

format-universal-time formats the time specified by *universal-time*. **format-decoded-time** formats the time specified by *seconds*, *minutes*, *hours*, *day*, *month*, and *year*. *Dest* is any destination accepted by the **format** function. The keyword arguments have the following meaning:

:timezone is an integer specifying the hours west of Greenwich. *:Timzone* defaults to the current time zone.

:style specifies the style to use in formating the time. The legal values are:

- *:short* specifies to use a numeric date.
- *:long* specifies to format months and weekdays as words instead of numbers.
- *:abbreviated* is similar to long except the words are abbreviated.
- *:government* is similar to abbreviated, except the date is of the form "day month year" instead of "month day, year".

:date-first if non-**nil** (default) will place the date first. Otherwise, the time is placed first.

:print-seconds if non-**nil** (default) will format the seconds as part of the time. Otherwise, the seconds will be omitted.

:print-meridan if non-**nil** (default) will format "AM" or "PM" as part of the time. Otherwise, the "AM" or "PM" will be omitted.

:print-timezone if non-**nil** (default) will format the time zone as part of the time. Otherwise, the time zone will be omitted.

:print-seconds if non-**nil** (default) will format the seconds as part of the time. Otherwise, the seconds will be omitted.

:print-weekday if non-**nil** (default) will format the weekday as part of date. Otherwise, the weekday will be omitted.

3.6. Lisp Library

The CMU Common Lisp project maintains a collection of useful or interesting programs written by users of our system. The library is in "**/afs/cs/project/clisp/library/**". There are two files there users should read:

CATALOG.TXT This file contains a page for each entry in the library. It contains information such as the author, portability or dependency issues, how to load the entry, etc.

READ-ME.TXT This file describes the library's organization and all the possible pieces of information an entry's catalog description could contain.

Hemlock has a command "**Library Entry**" that displays a list of the current library entries in an editor buffer. There are mode specific commands that display catalog descriptions and load entries. This is a simple and convenient way to browse the library.

Chapter 4

Error System

Written by Kent M. Pitman and Bill Chiles

4.1. Introduction

This chapter describes the Common Lisp Condition System, as proposed and accept by the X3J13 subcommittee on error handling. The design is primarily fixed, but the standards committee is still making changes to complete and polish it. Most of the work that remains is fully specifying the standard condition types (described below) and which Common Lisp functions must signal what conditions under what situations. CMU Common Lisp defines the conditions specified near the end of this chapter, but it does not signal all of them when you might expect. Therefore, we support a somewhat unsophisticated environment for extremely clever condition handling; however, you will probably find more functionality and conditions implemented than you'll need.

4.1.1. Purpose

Often we find it useful to describe a function in terms of its behavior in *normal situations*. For example, we may say informally that the function + returns the sum of its arguments or that the function `read-char` returns the next available character on a given input stream. Sometimes *exceptional situations* arise which do not fit neatly into such descriptions. For example, + might receive an argument which is not a number, or `read-char` might receive a single argument which was a stream that had no more available characters. This distinction between normal and exceptional situations is in some sense arbitrary, but is often very useful in practice.

For example, suppose you had a function `F` which you defined to allow only integer arguments, but you also guaranteed that the function `F` would detect and signal an error for non-integer arguments. Such a description is in fact internally inconsistent because the behavior is well-defined for non-integers. Yet we would not want this to force us to have to describe `F` as a function that accepts any kind of argument (just in case someone calls `F` only as a quick way to signal an error, for example). Using our new terminology, we can say clearly that `F` accepts integers in the normal situation, and signals an error in exceptional situations. Moreover, we can say that when we refer to the definition of a function informally, it is acceptable to speak only of its normal behavior. For example, we can speak informally about `F` as a function that accepts only integers without feeling that we are committing some fraud.

Not all exceptional situations are errors. For example, a program which is typing out a long line of text may notice that it is at the end of the line. It is possible that no real harm will result from continuing to type past the end of the line because the operating system will simply force a carriage return on the output device and continue typing on the next line. Even though the system recovers, it may be interesting to establish a protocol whereby that program can inform its callers of end-of-line exceptions. The controlling program could then opt to deal with these situations in interesting ways at certain times. It might choose to terminate printing, obtaining an end-of-line truncation. The point is the printer program can continue to operate correctly even when the controlling program

fails to provide advice about the situation; the situation is not an error.

Mechanisms for dealing with exceptional situations vary widely. When one occurs, a program may attempt to handle the situation by returning a distinguished value, returning an additional value, setting a variable, calling a function, performing a special transfer of control, or by stopping the program altogether and entering the debugger. For the most part, the facilities described in this document do not introduce any fundamentally new way of dealing with exceptional situations; rather, they encapsulate and formalize useful patterns of data and control flow which programmers have found useful in dealing with exceptional situations.

A proper conceptual approach to errors should begin with a discussion of the principles of *conditions* in general and eventually work its way up to the concept of an *error* as just one of the many kinds of conditions. However, given the widespread primitive state of error handling technology, a proper buildup may be as inappropriate as requiring that a beggar learn to cook a gourmet meal before being allowed to eat. As such, this chapter will first deal with the essentials, error handling, and then later go back and fill in the missing details.

4.1.2. Terminology

Common Lisp: the Language (pp. 5-6) says the following about errors:

> When this manual specifies that it "is an error" for some situation to occur, this means that:
>
> - No valid Common Lisp program should cause this situation to occur.
> - If the situation occurs, the effects are completely undefined as far as adherence to the Common Lisp specification is concerned.
> - No Common Lisp implementation is required to detect such an error. Of course, implementors are encouraged to provide for detection of such errors wherever reasonable.
>
> This is not to say that some particular implementation might not define the effects and results for such a situation; the point is that no program conforming to the Common Lisp specification may correctly depend on such effects or results.
>
> On the other hand, if it is specified in this manual that in some situation "an error is *signalled*", this means that:
>
> - If this situation occurs, an error will be signalled (see `error` and `cerror`).
> - Valid Common Lisp programs may rely on the fact that an error will be signalled.
> - Every Common Lisp implementation is required to detect such an error.
>
> In places where it is stated that so-and-so "must" or "must not" or "may not" be the case, then it "is an error" if the stated requirement is not met. For example, if an argument "must be a symbol", then it "is an error" if the argument is not a symbol. In all cases where an error is to be *signalled*, the word "signalled" is always used explicitly in this manual.

This has changed for the language standard. We have some new terms and phrases for describing what built in functions do. A *condition* is an interesting situation in a program which has been detected and announced. Later we will allow this term to also refer to objects which programs use to represent such situations. An *error* is a condition in which normal program execution may not continue without some form of intervention, either interactively by the user or under some sort of program control as described later in this document. *Signalling* is the process by which a program formally announces a condition. The `signal` function is the primitive mechanism that makes such announcements. Other abstractions, such as `error` and `cerror`, are built using `signal`.

Common Lisp: the Language is ambiguous about the reason why a particular program action is an error. There are two principal reasons why an action may be an error without requiring the signalling of an error:

- Detecting the error might be prohibitively expensive. Consider the following example:

  ```
  (+ nil 3)
  ```

  This is an error. It is likely that the designers of Common Lisp believed that this would be an error in all implementations but that they felt that it might be excessively expensive to detect the problem in

compiled code on stock hardware, so they did not require that + signal an error.

- Some implementations might implement the behavior as an extension. Consider the following example:

```
(loop for x from 1 to 3 do (print x))
```

This is an error because `loop` is not defined to take atoms in its body. Some implementations offer an extension which makes this well-defined. In order to leave room for such extensions, *Common Lisp: the Language* uses the "is an error" terminology to keep implementors from being forced to signal an error in the extended implementations.

This chapter uses the following terminology, which has been accepted by the X3J13's error handling subcommittee to become the standard in the next edition of the Common Lisp specification:

- If the signalling of a condition or error is part of a function's contract for specified situations, this documentation will say that it "signals" or "must signal" that condition or error.
- If the signalling of a condition or error is optional for some important reason, such as performance, this documentation will say that the program "might signal" that condition or error. In this case, it defines the operation to be illegal in all implementations, but allowing some implementations to avoid actually detecting the error.
- If an action is left undefined for the sake of an implementation-dependent extension, this documentation will say that it "is undefined" or "has undefined effect." This means that it is not possible to portably depend upon the effects of that action. A program which has an undefined effect could do anything including entering the debugger, transfering control, or modifying data in unpredictable ways.
- In the special case where only the return value of an operation is undefined, but any side-effect and transfer-of-control behavior is well defined, this documentation will say that it has "undefined value." In this case, the number and nature of the return values is undefined, but the user can reasonably expect the function to return. Under this description, there are some though not many, legitimate ways in which such return values can be used. For example, if the function `foo` has no side-effects and undefined value, the expression `(list (foo))` is completely well-defined even for portable code, but the effect of `(print (foo))` is not well-defined.

4.2. Concepts

4.2.1. Signalling Errors

Signalling an error in a program is an admission by that program that it does not know how to continue and requires external intervention. Once it signals an error, any decision about how to continue must come from outside of it.

The simplest way to signal an error is to use the `error` function with `format`-style arguments describing the error. If a piece of code calls `error`, and there are no active handlers (described later), the system enters the debugger and outputs the error message. For example, you might see an interaction such as the following:

```
* (defun factorial (x)
    (cond ((or (not (typep x 'integer)) (minusp x))
           (error "~S is not a valid argument to FACTORIAL." x))
          ((zerop x) 1)
          (t (* x (factorial (1- x))))))
FACTORIAL
* (factorial 20)
2432902008176640000
* (factorial -1)

Error in function FACTORIAL.
-1 is not a valid argument to FACTORIAL.

Restarts:
  0: Return to Top-Level.

Debug  (type H for help)
(FACTORIAL -1)
0]
```

A call to **error** cannot directly return. Unless a program prepares for special flow of control to override this behavior, **error** enters the debugger, and there will be no option to continue. An implementation's debugger may provide commands for interactively returning from individual stack frames, but the user must know what he is doing. The point is programs are written as if **error** never returns even though an implementation's environment may provide special development features.

A programmer may have a single, well-defined idea of a recovery strategy for an error, in which case he can use the function **cerror**. This specifies information to the user about what would happen if the user does continue from the call to **cerror**. For example:

```
* (defun factorial (x)
    (cond ((not (integerp x))
           (error "~s is not a valid argument to FACTORIAL." x))
          ((minusp x)
           (let ((x-magnitude (- x)))
             (cerror "Compute -(~D!) instead."
                     "(-~D)! is not defined."
                     x-magnitude)
             (- (factorial x-magnitude))))
          ((zerop x) 1)
          (t (* x (factorial (- x 1))))))
FACTORIAL
* (factorial -3)

Error in function FACTORIAL.
(-3)! is not defined.

Restarts:
  0: Compute -(3!) instead.
  1: Return to Top-Level.

Debug  (type H for help)
(FACTORIAL -1)
0] restart 0
-6
```

4.2.2. Trapping Errors

By default, **error** enters the debugger. You can override this behavior in a variety of ways; the simplest and most general mechanism is to wrap your code in an **ignore-errors** form. Normally forms in the body of **ignore-errors** evaluate sequentially returning the last value. If the evaluation of this code results in the signalling of a condition of type **error**, **ignore-errors** immediately returns two values: **nil** and the signalled condition object (described later). The system does not invoke the debugger or print any error messages. For example:

```
* (setq filename "nosuchfile")
"nosuchfile"
* (ignore-errors (open filename :direction :input))
NIL
#<FILE-ERROR.5EA4>
```

Usually, **ignore-errors** is undesirable because it handles every possible kind of error. Though some may argue differently, a program which avoids entering the debugger is not necessarily better than one which does enter it. Excessive use of **ignore-errors** keeps the user out of the debugger, but it does little to increase your program's reliability; your program may continue running after encountering errors other than those you designed it to ignore. In general, it is better to deal with the particular errors that you believe could occur, and if an unexpected error does happen, you will find out about it.

The error system defines **ignore-errors** on a more general facility called **handler-case**. It allows the user to specifically deal with types of conditions, including non-**error** conditions, without affecting the signalling of disjoint or more general kinds of conditions. The following example achieves an equivalent effect to the previous example's use of **ignore-errors**:

```
* (setq filename "nosuchfile")
"nosuchfile"
* (handler-case (open filename :direction :input)
    (error (condition) (values nil condition)))
NIL
#<FILE-ERROR.5EA9>
```

The advantage of **handler-case** in this scenario is the ability to specify a more specific condition type than **error**. Condition types are explained in detail later, but the following should be a clear example:

```
* (makunbound 'filename)
FILENAME
* (handler-case (open filename :direction :input)
    (file-error (condition) (values nil condition)))

Error: The variable FILENAME is unbound.

Restarts:
  1: Retry getting the value of FILENAME.
  2: Specify a value of FILENAME to use this time.
  3: Specify a value of FILENAME to store and use.
  4: Return to Top-Level.

Debug  (type H for help)
0]
```

4.2.3. Handling Conditions

The basic idea of condition handling involves the signalling of a condition. A piece of code called the *signaller* recognizes and announces an exceptional situation using **signal**, or some function built on it such as **error**. The process of signalling includes the search for and invocation of a *handler*, a piece of code that will attempt to take care of the situation appropriately. If this process finds a handler, it may either *handle* the situation by performing

some non-local transfer of control, or it may *decline* by refusing to perform a non-local transfer of control. Whenever a handler declines, the search for a willing handler continues.

Since the lexical environment of the signaller might not be available to handlers, the system supports a data structure called a *condition* to represent the relevant state of the situation. Users can also create conditions explicitly using **make-condition** and pass them to a function such as **signal**, or they can allow the system to create conditions implicitly by using functions such as **signal** and **error**. To handle a condition a handler can use any non-local transfer of control including the following:

- go to a tag in a **tagbody**
- **return** from a **block**
- **throw** to a **catch**

The system provides abstractions built on these primitives for convenience in exception handling. For example, **handler-bind** makes a handler dynamically accessible to a program, and the following creates a handler for a condition of type **arithmetic-error**:

```
(handler-bind ((arithmetic-error #'this-handler))
  ...body...)
```

A handler is a function of one argument, a condition. While **body** executes, if someone signals a condition of the designated type, and there are no dynamically intervening handlers, **signal** invokes the handler on the given condition. The following is a complete example showing a macro that handles **arithmetic-error**'s by returning **nil** and the condition if the arithmetic could not be computed:

```
(defmacro without-arithmetic-errors (&body forms)
  (let ((tag (gensym)))
    `(block ,tag
       (handler-bind ((arithmetic-error
                       #'(lambda (condition)
                           (return-from ,tag
                                        (values nil condition)))))
         ,&body))))
```

Handlers execute in the dynamic context of the signaller, but the system rebinds the set of available condition handlers to those that were active at the time the program established the handler. This means that if the handler signals a condition or calls something that signals one, the handler and any others bound in the same **handler-bind** form are inaccessible to the signalling process.

If the system can only find handlers that decline, and the condition is signalled via **error** or **cerror**, or similar routines, the system enters the debugger within the dynamic context of the signaller.

4.2.4. Object-Oriented Basis of Condition Handling

The ability of the handler to usefully handle an exceptional situation is related to the quality of the information given to it. If we only signalled errors with a string describing the condition, **string-equal** would be a handler's best tool for identifying what happened, and the information presented to the user would be the same as the string passed to the handler. It would be ridiculous to try to map what was passed to the error system to something different to display to the user.

It is fundamentally important to decouple the error message string from the objects which formally represent the error state. Thus, there is a notion of typed conditions and formal operations on them which make them inspectable in a structured way. This object-oriented approach to condition handling has the following important advantages over a text-based approach:

- Conditions are classified according to subtype relationships, making it easy to test for categories of conditions.

- Conditions have named slot values through which parameters are conveyed from the program that signals the condition to the program that handles it.
- Inheritance of methods (in a loose sense) and slots reduce the amount of explicit specification necessary to achieve various interesting effects.

This document describes some predefined condition types, and the set of condition types is extensible using **define-condition**. The following is an example defining a function of two arguements called **divide** that is patterned after the / function and that does some error checking:

```
(defun divide (numerator denominator)
  (cond ((or (not (numberp numerator)) (not (numberp denominator)))
         (error "(DIVIDE '~S '~S) - Bad arguments."
                numerator denominator))
        ((zerop denominator)
         (error 'division-by-zero
                :operator 'divide
                :operands (list numerator denominator)))
        (t ...)))
```

In the first clause, the definition uses **error** with a string argument, and in the second clause it names a particular condition type, **division-by-zero**. In the case of a string argument, the system signals a **simple-error** condition type.

The particular kind of error signalled may be important when handlers are actually active. For example, **simple-error** inherits from type **error**, which in turn inherits from type **condition**. In the other case, **division-by-zero** inherits from **arithmetic-error**, which inherits from **error**, etc. If a handler existed for **arithmetic-error** when some code signals a **division-by-zero** condition, the system would invoke that handler; however, if the same code in the same context signals a **simple-error** condition, the system would ignore the handler for the **arithmetic-error** type.

4.2.5. Restarts

The condition system separates the act of signalling an error of a particular type from the means of recovering from that error in some way. In the **divide** example in the previous section, signalling an error does not imply a willingness on the part of the signaller to cooperate in any corrective action. For example, if the user ends up in the debugger, his only option may be to return to the Lisp top level.

When a program detects an error and calls **error**, execution cannot continue normally because **error** never returns directly. However, the user can write his program to transfer control to other points in the program with specially established *restarts*. The simplest restart involves structured transfer of control using a macro called **restart-case**. The **restart-case** form allows the programmer to execute a piece of code in a context where zero or more restarts are active, and if the program or the user, through the debugger, invokes one of those restarts, the system transfers control to the corresponding clause in the **restart-case** form.

The following shows the **divide** example from the previous section rewritten:

```
(defun divide (numerator denominator)
  (loop
    (restart-case (return
                    (cond ((or (not (numberp numerator))
                               (not (numberp denominator)))
                           (error "(DIVIDE '~S '~S) - Bad arguments."
                                  numerator denominator))
                          ((zerop denominator)
                           (error 'division-by-zero
                                  :operator 'divide
                                  :operands
                                  (list numerator denominator)))
                          (t ...)))
      (nil (arg1 arg2)
           :report
           "Provide new arguments for use by the DIVIDE function."
           :interactive (lambda ()
                          (list (prompt-for 'number "Numerator: ")
                                (prompt-for 'number "Denominator: ")))
           (setq numerator arg1 denominator arg2))
      (nil (result)
           :report
           "Provide a value to return from the DIVIDE function."
           :interactive
           (lambda () (list (prompt-for 'number "Result: ")))
           (return result)))))
```

The **nil** at the head of each clause means that it is an *anonymous* restart. Anonymous restarts are typically only invoked from within the debugger. Later sections describe in detail *named* restarts that programs can call, typically from handlers, without the need for user intervention. If the arguments to anonymous restarts are required, not optional, the code must specify the **:interactive** keyword to provide information concerning how to supply the arguments in case the user causes its invocation via the debugger.

The **:report** keyword specifies how to present the restart option to the user, such as in the debugger.

In this example, **prompt-for** is immaterial and does what you think.

The following is a sample interaction that takes advantage of the restarts provided by the revised definition of **divide**:

```
* (+ (divide 3 0) 7)

Error in function DIVIDE.
Attempt to divide 3 by 0.

Restarts:
  0: Provide new arguments for use by the DIVIDE function.
  1: Provide a value to return from the DIVIDE function.
  2: Return to Top-Level.

Debug  (type H for help)
(DIVIDE 3 0)
0] restart 0
Numerator: 4
Denominator: 2
9
```

4.2.6. Named Restarts

Named restarts are more powerful or convenient than unnamed ones since programs and users can invoke them without the aid of an interface like the debugger. The following is a degenerate, interesting example:

```
(restart-case (invoke-restart 'foo 3)
  (foo (x) (+ x 1)))
```

This adds 3 to 1, returning 4, and it is analagous to writing:

```
(+ (catch 'something (throw 'something 3)) 1)
```

A more practical example below shows a possible portion of Lisp's **symbol-value** function that signals an **unbound-variable** error:

```
(restart-case (error 'unbound-variable :name variable)
  (continue ()
    :report (lambda (stream)
              (format stream "Retry getting the value of ~S."
                      variable))
    (symbol-value variable))
  (use-value (value)
    :report (lambda (stream)
              (format stream "Specify a value of ~S to use this time."
                      variable))
    value)
  (store-value (value)
    :report (lambda (stream)
              (format stream "Specify a value of ~S to store and use."
                      variable))
    (setf (symbol-value variable) value)
    value))
```

With this, users can write a variety of automatic handlers for **unbound-variable** errors. The following makes unbound variables evaluate to themselves:

```
(handler-bind ((unbound-variable
                #'(lambda (condition)
                    (if (find-restart 'use-value)
                        (invoke-restart
                         'use-value
                         (cell-error-name condition))))))
  ...body...)
```

4.2.7. Restart Functions

Some restarts or recovery techniques are common in that programmers find themselves writing very similar restart cases. It is good style to provide a simpler invocation means than is otherwise used for these. *Restart functions* hide the typical use of **invoke-restart**.

Conventionally the restart function shares the name of the restart name. The system defined functions **abort**, **continue**, **muffle-warning**, **store-value**, and **use-value** are restart functions. With **use-value**, the **handler-bind** example at the end of the previous section that handles **unbound-variable** errors becomes much simpler:

```
(handler-bind ((unbound-variable
                #'(lambda (condition)
                    (use-value (cell-error-name condition)))))
  ...body...)
```

Textually the example only saves two lines of code, but conceptually the handler doesn't have to be concerned with whether the restart is currently active. You don't want your handler to get an unactive restart error because you forgot to make sure the restart exists. **Use-value** takes care of that and simply returns if the restart is unactive,

causing the handler to return indicating that it declines handling the condition.

4.2.8. Contrasting Restarts and Catch/Throw

One important feature **restart-case** offers which **catch/throw** does not is the ability to reason about the available points to which a program transfers control without actually attempting the transfer. Considering the following, the first form is a poor man's variation of the second:

```
(ignore-errors (throw ...))

(if (find-restart 'something) (invoke-restart 'something))
```

The following two forms are much cleaner than the programming required using **ignore-errors**, **throw**, and hacks with binding specials to know what context you are in:

```
(if (and (find-restart 'something) (find-restart 'something-else))
    (invoke-restart 'something))

(if (and (find-restart 'something) (yes-or-no-p "Do something? "))
    (invoke-restart 'something))
```

Simply using **ignore-errors** and **throw** forces a transfer of control when it possibly is inconvenient or an error, and the restart mechanism readily provides a means for inspecting the dynamic context and interacting with the user.

Another difference between the restart facility and the **catch/throw** facility is that a **catch** with any given tag completely shadows any outer, pending **catch** with the same tag. Because of the **compute-restarts** function, it is possible to see shadowed restarts which can be very useful, such as in the debugger.

4.2.9. Generalized Restarts

Restart-case allows only imperative transfer of control for its associated restarts. The system defines it using a lower level primitive called **restart-bind** which does not force transfer of control. Its syntax is as follows:

```
(restart-bind ((name function . options)) . body)
```

body executes in a dynamic context where **(invoke-restart 'name)** invokes **function**. The **options** are keyword-style and describe information similar to that provided with the **:report** keyword in **restart-case**. A **restart-case** expands into a call to **restart-bind** with functions that unconditionally transfer control to a particular body of code, passing along arguments.

Restarts can be useful without transfering control. Consider the following example:

```
(restart-bind ((nil #'(lambda () (expunge-directory the-dir))
                    :report-function
                    #'(lambda (stream)
                        (format stream "Expunge ~A."
                                (directory-namestring the-dir)))))
  (cerror "Try this file operation again."
          'directory-full :directory the-dir))
```

Entering the debugger in this context, the user could perform the expunge, avoiding transfering control from within the debug context, and then retry the file operation, as in:

```
* (open "foo" :direction :output)

Error in function OPEN.
The directory /usr/bovik/ is full.

Restarts
  0: Try this file operation again.
  1: Expunge /usr/bovik/.
  2: Return to Lisp Top-Level.

Debug  (type H for help)
(OPEN "foo" :DIRECTION :OUTPUT)
0] restart 1
Expunging /usr/bovik/ ...
0] restart 0
#<File stream "/usr/bovik/foo">
```

4.2.10. Serious Conditions

The **ignore-errors** macro will trap conditions of type **error**, but since this form is so dangerous for squelching every kind of error, some conditions are very serious without being a subtype of **error**. These are of type **serious-condition**, and the system might use this type for situations such as stack overflow or exhausted storage. The type **error** is a subtype of **serious-condition**, and though it is technically correct to refer to errors as *serious conditions*, we typically reserve that phrase to indicate conditions that are subtypes of **serious-condition** excluding subtypes of **error**.

This distinction is necessary to provide for exceptions that don't fall under the domain of the Common Lisp language. We assume an implementation uses a stack for function calling, and we know that stacks can overflow; however, this is not a programming error. In another implementation, the same program might run fine. Furthermore, if a program is dynamically within an **ignore-errors** form, and the system runs out of memory or the stack overflows, the system must stop or take care of this. The conditions simply cannot be ignored.

By convention, programmers prefer the function **error** over **signal** to signal conditions of type **serious-condition**, as well as those of type **error**. It is the use of the function **error**, and not the type of the signalled condition, that causes the system to enter the debugger.

4.2.11. Non-Serious Conditions

Some conditions are neither errors nor serious conditions. Programs signal these to give other programs a chance to intervene, but if none take any action, then computation simply continues normally. For example, an implementation might choose to signal a non-serious condition called **end-of-line** when output reaches the last character position on a line of character output. In such an implementation, the signalling of this condition allows a convenient way for other programs to do something special in this situation, producing output that is truncated at the end of a line or simulating a line-wrapping device.

Use **signal** to signal these types of conditions. If the program uses **error** to signal a non-serious condition, the system will still enter the debugger if it goes unhandled. The point of signalling a non-serious condition is that it should not matter if a program continues to execute immediately after the signalling regardless of whether some other program took any action based on the situation.

4.2.12. Condition Types

Some types of conditions are predefined by the system. All types of conditions are subtypes of **condition**. That is, **(typep c 'condition)** is true if and only if **c** is a **condition** object.

Implementations supporting multiple (or non-hierarchical) type inheritance are expressly permitted to exploit multiple inheritance in the tree of condition types as implementation-dependent extensions, as long as such extensions are compatible with this document.

In order to avoid problems in portable code which run both in systems with multiple type inheritance and systems without it, the designers warn against assuming subtype relationships specified in this document are mutually exclusive. In some cases this document does specify disjoint subtypes, but this is not the default. For example, from the subtype descriptions contained in this document, in all implementations the following must be true:

```
(typep c 'control-error) implies (typep c 'error),
```

However, the reader must avoid the following assumption:

```
(typep c 'control-error) implies (not (typep c 'cell-error))
```

4.2.13. Signalling Conditions

When a program signals a condition, the system tries to locate the most appropriate handler for the condition and invoke that handler. There are constructs for dynamically establishing handlers. If the process of signalling finds a suitable handler, it calls the handler. Sometimes handlers decline by simply returning without performing a non-local transfer of control. When this happens, the search for an appropriate handler continues as if the handler never existed. When **signal** fails to find a handler to take care of the situation, it returns **nil**.

It is worth noting the handler search procedure finds dynamically more local handlers before it finds those established dynamically earlier in time, regardless of whether the more local handler is more specific than any earlier bound handler. Therefore, the programmer should take care when binding handlers to very general condition types since a more specific handler may already be established that is more appropriate. There is no reason to be overly concerned about this, experience with existing condition systems suggests that this is a reasonable approach and works adequately in most situations.

4.2.14. Condition Handlers

A *handler* is a function of one argument, the signalled condition. The handler may inspect the object to see if it really wants to take care of it. Handlers execute in the dynamic context of the signaller, but the system rebinds the set of available condition handlers to those that were active at the time the program established the handler. The intent of this is to prevent infinite recursion due to errors in a condition handler.

After inspecting the condition, the handler should take one of the following actions:

- *Decline* to handle the condition by simply returning.
- Handle the condition by performing some non-local transfer of control. This may be done either primitively using **go**, **return**, and **throw** or more abstractly using a function such as **abort** or **invoke-restart**.
- Signal another condition.
- Invoke the interactive debugger.

The latter two items are really ways of putting off the decision to either handle or decline, in case some other code or the user wants to get in on the recovery action. Ultimately, all a handler can do is handle or decline to handle a condition.

4.2.15. Printing Conditions

When ***print-escape*** is **nil**, as with **princ** or the **~A format** option, the system invokes the report method for the condition. This is the means for presenting conditions to users. Some functions, **invoke-debugger**, **break**, and **warn**, always display the condtion, but users can explicity cause conditions to

report themselves when desired:

```
(defun open-data-file (user-specified-name default-system-name)
  (handler-case (open user-specified-name)
    (serious-condition (condition)
      (format t "~&Opening ~S failed:~%~A~&Using ~S instead."
              user-specified-name condition default-system-name)
      (open default-system-name))))
```

This might print something like the following:

```
Opening #.(pathname "/usr/dat/unavailable-data-file") failed:
#.(pathname "/usr/dat/unavailable-data-file") is read protected.
Using #.(pathname "/usr/dat/default-objects") instead.
```

Some notes about the text presented by report methods:

- The message should be a complete sentence, beginning with a capital letter and ending with appropriate punctuation.
- The message should exclude introductory text such as "**Error:**" or "**Warning:**". Such text will be added by the routine invoking the report method as appropriate to the context.
- Except where unavoidable, tab characters should be avoided in error messages. Their effect can vary between implementations and can cause problems even within an implementation because it may output a variety of space depending on the current printing column when the condition reports.
- Single line messages are preferred, but newlines in the middle of long messages are acceptable.
- If any program displays messages indented from the prevailing left margin, possibly to make the report standout against other text, then that program will take care to insert the indentation into any extra lines of a multi-line error message. Similarly, any program that prefixes error messages with semicolons so that they appear to be comments should take care of inserting a semicolon at the beginning of each line in a multi-line error message.

When ***print-escape*** is non-**nil**, the object should print in some useful and fairly abbreviated fashion according to the style of the implementation. The condition may print unreadably, as by **read**; that is, it may use "#<" syntax.

4.3. Signalling Conditions

error *datum* **&rest** *arguments* [*Function*]

This function invokes the signal facility on a condition formed from *datum* and *arguments*. If the condition is unhandled, this calls **invoke-debugger** on the condition. This function never returns and can only be exited by a non-local transfer of control in a handler or by use of a debugger command.

This uses *datum* and *arguments* as follows:

- If *datum* is a condition, then this uses it directly. In this case, it is an error for *arguments* to be anything other than **nil**.
- If *datum* is a condition type, then this uses the condition resulting from **apply**'ing **make-condition** to *datum* and *arguments*.
- If *datum* is a string, then this uses the condition resulting from the following:

```
(make-condition 'simple-error
                :format-string datum
                :format-arguments arguments)
```

cerror *continue-format-string datum* **&rest** *arguments* [*Function*]

This function invokes the error facility on a condition formed from *datum* and *arguments*. If the condition is unhandled, this calls **invoke-debugger** on the condition. While signalling the condition, and while in the debugger if it is reached, it is possible to continue program execution using the **continue** restart.

This uses *datum* and *arguments* as follows:

- If *datum* is a condition, then this uses it directly. In this case, **cerror** only uses *arguments* with *continue-format-string*, and it will not use *arguments* to initialize *datum* in any way.
- If *datum* is a condition type, then this uses the condition resulting from **apply**'ing **make-condition** to *datum* and *arguments*. In this case, **cerror** uses *arguments* with *continue-format-string* in a call to **format** and with *datum* in a call to **make-condition**, so the user must take care to set up the **format** string correctly. The directive **~*** may be useful in this situation.
- If *datum* is a string, then this uses the condition resulting from the following:

```
(make-condition 'simple-error
                :format-string datum
                :format-arguments arguments)
```

continue-format-string must be a string, and **cerror** returns **nil**.

signal *datum* **&rest** *arguments* [*Function*]

break-on-signals [*Variable*]

This function invokes the signal facility on a condition formed from *datum* and *arguments*. If the condition is unhandled, **signal** returns **nil**.

This uses *datum* and *arguments* as follows:

- If *datum* is a condition, then this uses it directly. In this case, it is an error for *arguments* to be anything other than **nil**.
- If *datum* is a condition type, then this uses the condition resulting from **apply**'ing **make-condition** to *datum* and *arguments*.
- If *datum* is a string, then this uses the condition resulting from the following:

```
(make-condition 'simple-condition
                :format-string datum
                :format-arguments arguments)
```

If the following test is true, then this function enters the debugger before beginning the signalling process:

```
(typep condition *break-on-signals*)
```

The user can continue this invocation of the debugger using the **continue** restart. Note, this is true for functions and macros that use **signal**: **error**, **cerror**, **warn**, **assert**, and **check-type**.

The condition system provides ***break-on-signals*** for debugging programs that do signalling. The user should choose the most restrictive specification that suffices. Setting this flag effectively violates the modular handling of condition signalling this system seeks to establish, and the effect may be unpredictable in some cases since the user may not be aware of the variety or number of calls to **signal** large sophisticated programs use.

warn *datum* **&rest** *arguments* [*Function*]

break-on-warnings [*Variable*]

This function warns about a situation by signalling a condition of type **warning** formed from *datum* and *arguments*.

This uses *datum* and *arguments* as follows:

- If *datum* is a condition, this uses condition directly. In this case, if the condition is not of type `warning`, or *arguments* is something other than `nil`, `warn` signals an error of type `type-error`.
- If *datum* is a condition type, then this uses the condition resulting from `apply`'ing `make-condition` to *datum* and *arguments*. If this is anything other than a subtype of `warning`, `warn` signals a `type-error` error.
- If *datum* is a string, then this uses the condition resulting from the following:

```
(make-condition 'simple-warning
                :format-string datum
                :format-arguments arguments)
```

If `*break-on-warnings*` is true, then the `warn` enters the debugger using `break` before signalling the `warning` condition. Because of the use of `break`, the `continue` restart allows `warn` to continue executing normally. This feature is only supported for compatibility with previous condition system proposals that some implementations implemented. The `*break-on-signals*` mechanism supersedes `*break-on-warnings*` and is more general in comparison. Programmers should write new code using the `*break-on-signals*`, and if they want to break on warnings, then they should set the variable to `'warning`. The condition system provides these features for debugging programs that issue warnings.

The precise mechanism for warning is as follows:

1. If `*break-on-warnings*` is true, `warn` calls `break`. If the user continues the break the `continue` restart, proceed with step 2.
2. Signal the `warning` condition with an active `muffle-warning` established. This allows handlers to cause `warn` to immediately return `nil` without taking any other action. If the condition goes unhandled, proceed with step 3.
3. Report the condition to `*error-output*`.
4. Return `nil`.

4.4. Handling Conditions

`handler-case` *form* {*case*}* [*Macro*]

This macro executes *form* in a context where various handlers are active as specified by each *case*. Each *case* is of the following form:

```
(type ([var]) . body)
```

Type may be any type specifier. If during the execution of *form*, the code signals a condition for which there is an appropriate clause (that is, the condition's type is a subtype of one of the specified types), and there is no intervening handler for the condition's type, then the system transfers control to the body of the clause. The code in the clause executes in the dynamic context of the signaller, but with respect to bound handlers, the system alters the context to that immediately prior to establishing the invoked handler. The code executes also with *var* bound to the signalled condition. If *form* runs to a normal completion, then `handler-case` returns the values resulting from it.

If *var* is unneeded, it may be omitted. For example, a clause such as:

```
(type (var) (declare (ignore var)) form)
```

may be written more easily in the following way:

```
(type () form)
```

A clause may have no forms after the argument specification. In this case, if the system transfers control to this clause, it returns `nil`.

The signalling process searches the clauses from top to bottom, as if the textually earlier clauses were dynamically bound later in time than the textually later clauses. This is analogous to **typecase**. If there is a type overlap, **signal** transfers control to the textually first case by way of invoking the handler. For purposes of invoking any one handler case, the dynamic context of bound handlers excludes all handlers established by a single **handler-case**.

As a special case, the *type* can be the symbol **:no-error** in the last clause. If the user specifies this, it designates a case that executes if *form* returns normally, and the arguments passed to the case are those returned by *form*.

Examples of **handler-case**:

```
(handler-case (/ x y)
  (division-by-zero () nil))

(handler-case (open *the-file* :direction :input)
  (file-error (condition)
    (format t "~&Fooey: ~A~%" condition)
    nil))

(handler-case (some-user-function)
  (file-error (condition) condition)
  (division-by-zero () 0)
  ((or unbound-variable undefined-function) () 'unbound))

(handler-case (intern x y)
  (error (condition) condition)
  (:no-error (symbol status)
             (declare (ignore symbol))
             status))
```

ignore-errors {*forms*}* [*Macro*]

This macro executes *forms* in a context that handles conditions of type **error** by returning from this form two values: **nil** and the signaled condition object. If the system does not invoke this handler, either because no one signaled an **error** condition or because a tighter bound handler took care of the error, **ignore-errors** returns any values returned by the last form executed.

This is equivalent to the following:

```
(handler-case (progn forms)
  (error (condition) (values nil condition)))
```

handler-bind ({(*type handler*)}*) {*form*}* [*Macro*]

This macro executes its body in a dynamic context where the given handler bindings are in effect. *Type* may be any type specifier. *Handler* should evaluate to a function used to handle conditions of the associated type(s) during execution of the body. This function takes a required argument that is the signaled condition.

The signalling process searches the bindings from top to bottom, as if the textually earlier bindings were dynamically bound later in time than the textually later bindings. This is consistent with **handler-case** which is analogous to **typecase**. If there is a type overlap, **signal** finds the earlier binding first. For purposes of invoking any one handler, the dynamic context of bound handlers excludes all handlers established by a single **handler-bind**.

If the body executes normally, this returns the values of the last *form*.

4.5. Defining and Creating Conditions

define-condition *name* (*parent-type*) [({*slot*}*) {*option*}*] [*Macro*]

This macro defines a new condition type called *name*, which is a subtype of the given *parent-type*. Except as otherwise noted, the expansion of this macro does not evaluate the arguments.

Objects of this condition type include slots available in objects of *parent-type* in addition to the indicated slots. A *slot* description has the following form:

{ *slot-name* | (*slot-name*) | (*slot-name default-value*) }

The *default-value* is a form evaluated by **make-condition** to produce a default value when the caller leaves the value unspecified. If the description of the slot does not provide a *default-value*, and the user of **make-condition** does not specify a value, then the system initializes the slot in an implementation-dependent way. It is an error to attempt to access a slot which has not been explicitly initialized and which has not been given a default value.

If the new type and some other type from which it inherits have a slot with the same name, **define-condition** only allocates one slot for the new type. Any specified default overrides any inherited default.

make-condition accepts keywords (in the keyword package) with the same name as any slot name and initializes the corresponding slot in conditions it creates.

Accessors are created according to the same rules used by **defstruct**, but it is an error to attempt to assign a condition's slots with **setf**. **Define-condition** interns accessor names into the package that is current when it executes.

A valid *option* is one of the following:

(:documentation *doc-string*)
: *Doc-string* is a string describing the purpose of the condition type or **nil**. If this option is unspecified, **define-condition** assumes **nil**. The documentation is retrievable with **(documentation name 'type)**, where **type** is the condition *name*.

(:conc-name *symbol-or-string*)
: As with **defstruct**, this sets up automatic prefixing of the names of slot accessors. The default is to use the name of the new type, *name*, followed by a hyphen.

(:report *exp*)
: If *exp* is not a literal string, it must be a suitable argument to the **function** special form, and when **define-condition** expands, it evaluates the expression **(function** *exp*) in the current lexical environment. The function takes two required arguments, a condition and a stream, and the system invokes the function whenever it prints the condition with ***print-escape*** bound to **nil**. See section 4.2.15. If *exp* is a literal string, it is a shorthand for the following:

```
(lambda (condition stream)
  (declare (ignore condition))
  (write-string exp stream))
```

This option inherits from *parent-type* if not specified.

Here are some examples of defining conditions. This form defines a condition type called **machine-error** which inherits from type **error**:

```
(define-condition machine-error (error)
  (machine-name)
  (:report (lambda (condition stream)
             (format stream "There is a problem with ~A."
                     (machine-error-machine-name condition)))))
```

The slot **machine-name** can be accessed with **machine-error-machine-name**, and

make-condition will accept a **:machine-name** keyword when creating conditions of type **machine-error**.

This defines a condition subtype of **machine-error** to be used when machines are not available:

```
(define-condition machine-unavailable (machine-error)
  ()
  (:report (lambda (condition stream)
             (format stream "The machine ~A is not available."
                     (machine-error-machine-name condition)))))
```

The previous comments concerning **machine-error** apply to **machine-unavailable** conditions, and **machine-unavailable-machine-name** will also access the name of the problem machine.

This defines a still more specific condition type, a subtype of **machine-unavailable**, which provides a default for the **machine-name** slot:

```
(define-condition central-file-server-unavailable
                  (machine-unavailable)
  ((machine-name "cfs.cs.cmu.edu")))
```

Since this example leaves the **:report** option unspecified, it inherits the report method for **machine-unavailable** conditions.

make-condition *type* **&rest** *slot-initializations* [*Function*]

This function constructs a condition object of type *type* using *slot-initializations*. This returns the condition object. *Slot-initializations* is given as alternating keyword/value pairs. The following example shows the creation of a condition type **peg/hole-mismatch** with slots named **peg-shape** and **hole-shape**:

```
(make-condition 'peg/hole-mismatch
                :peg-shape 'square :hole-shape 'round)
```

4.6. Assertions

check-type *place typespec* **&optional** *string* [*Macro*]

This macro signals an error of type **type-error** if the contents of *place* are not of type *typespec*. If this signals a condition, handlers can use the functions **type-error-object** and **type-error-expected-type** to access the contents of *place* the desired type *typespec*, respectively. This form only returns if a handler, or the user from the debugger, invokes the **store-value** restart.

In this situation **store-value** takes an argument or prompts the user for it and stores the value in *place*, continuing within **check-type** which starts over possibly signalling an error again. Subforms of *place* may evaluate multiple times because of the implicit loop generated. **Check-type** returns **nil**.

Place must be a generalized variable reference acceptable to **setf**. *Typespec* must be a type specifier, and **check-type** does not evaluate it. *String* is a string literal describing type, and if it is unsupplied, **check-type** computes a description from *typespec*.

The error message will mention *place*, its contents, and the desired type.

Here are a couple examples:

```
* (setf aardvarks '(sam harry fred))
(SAM HARRY FRED)
* (check-type aardvarks (array * (3)))

The value of AARDVARKS is (SAM HARRY FRED),
 which is not of type (ARRAY * (3)).

Restarts:
  0: Supply a new value of AARDVARKS.
  1: Return to Top-Level.

Debug  (type H for help)
(LISP::CHECK-TYPE-ERROR AARDVARKS (SAM HARRY FRED)
                        (ARRAY * (3)) NIL)
0] restart 0
Type a form to be evaluated:
'#(sam fred harry)
NIL
* aardvarks
#(SAM FRED HARRY)
* (setf count 'foo)
FOO
* (check-type count (integer 0 *) "a positive integer")

The value of COUNT is FOO, which is not a positive integer.

Restarts:
  0: Supply a new value of COUNT.
  1: Return to Top-Level.

Debug  (type H for help)
(LISP::CHECK-TYPE-ERROR COUNT FOO
                        (INTEGER 0 *) "a positive integer")
0] restart 1
*
```

assert *test-form* **&optional** ({*place*}*) *datum* {*argument*}* [*Macro*]

This macro signals an error if the value of *test-form* is **nil**. Using the the **continue** restart allows the user to alter the values of some variables. If **continue** does execute, **assert** starts over, evaluating *test-form* and possibly signalling an error again. **Assert** returns **nil**.

Test-form is any form. Each *place* must be a generalized variable reference acceptable to **setf**. **Assert** only evaluates subforms of each *place* if the **continue** restart runs, and it may re-evaluate them each time the assertion fails.

This only evaluates *datum* and each *argument* if it signals a condition, and it will re-evaluate them each time the assertion fails. **Assert** uses these parameters in the following way:

- If *datum* is a condition, this uses it directly. In this case, it is an error to specify any *argument*.
- If *datum* is a condition type, then this uses the condition resulting from **apply**'ing **make-condition** to *datum* and *arguments*.
- If *datum* is a string, then this uses the condition resulting from the following:

```
(make-condition 'simple-error
                :format-string datum
                :format-arguments arguments)
```

- If *datum* is unsupplied, then this uses a condition of type `simple-error` constructed with *test-form* as data, for example:

```
(make-condition 'simple-error
                :format-string "The assertion ~S failed."
                :format-arguments '(test-form))
```

Here is an example of `assert`:

```
* (setf x (make-array '(3 5) :initial-element 3))
#2A((3 3 3 3 3) (3 3 3 3 3) (3 3 3 3 3))
* (setf y (make-array '(3 5) :initial-element 7))
#2A((7 7 7 7 7) (7 7 7 7 7) (7 7 7 7 7))
* (defun matrix-multiply (a b)
    (let ((*print-array* nil))
      (assert (and (= (array-rank a) (array-rank b) 2)
                   (= (array-dimension a 1) (array-dimension b 0)))
              (a b)
              "Cannot multiply ~S by ~S." a b)
      (really-matrix-multiply a b)))
MATRIX-MULTIPLY
* (matrix-multiply x y)

Error in function LISP::ASSERT-ERROR.
Cannot multiply #<Array, rank 2 {B8}> by #<Array, rank 2 {D4}>.

Restarts:
  0: Retry assertion with new values for A, B.
  1: Return to Top-Level.

Debug  (type H for help)
(LISP::ASSERT-ERROR (AND (= (# #) (# #) 2) (= (# # #) (# # #)))
                    (A B)
                    "Cannot multiply ~S by ~S."
                    #<Array, rank 2>...)
0] restart 0
The old value of A is #<Array, rank 2>.
Do you want to supply a new value? y
Type a form to be evaluated:
x
The old value of B is #<Array, rank 2>.
Do you want to supply a new value? y
Type a form to be evaluated:
(make-array '(5 3) :initial-element 6)
#2A((54 54 54 54 54)
    (54 54 54 54 54)
    (54 54 54 54 54)
    (54 54 54 54 54)
    (54 54 54 54 54))
*
```

4.7. Case Forms

This section describes case forms similar to `case` and `typecase` that signal an error if no branch fires.

etypecase *keyform* { (*type* {*form*}*) }* [*Macro*]

This control construct is similar to **typecase**, but no explicit **otherwise** or **t** clause is permitted. If no clause fires, this signals an error of type **type-error** with a message constructed from the clauses. This error cannot be continued. To supply your own error message, use **typecase** with an **otherwise** clause containing a call to **error**. The name of this function stands for "exhaustive type case" or "error-checking type case."

Here is an example:

```
* (setq x 1/3)
1/3
* (etypecase x
    (integer (* x 4))
    (symbol  (symbol-value x)))

1/3 fell through ETYPECASE expression.
Wanted one of (SYMBOL INTEGER).

Restarts:
  0: Return to Top-Level.

Debug  (type H for help)
(LISP::%EVAL (ERROR 'CONDITIONS::CASE-FAILURE
                    :NAME 'ETYPECASE :DATUM ...))
0]
```

ctypecase *keyplace* { (*type* {*form*}*) }* [*Function*]

This control construct is similar to **typecase**, but no explicit **otherwise** or **t** clause is permitted. The *keyplace* must be a generalized variable reference acceptable to **setf**. If no clause fires, **ctypecase** signals an error of type **type-error** with a message constructed from the clauses. The user may continue this error using the **store-value** restart.

In this situation **store-value** takes an argument or prompts the user for it and stores the value in *keyplace*, continuing within **ctypecase** which starts over possibly signalling an error again. Subforms of *keyplace* may evaluate multiple times because of the implicit loop generated.

This returns any values returned by the last *form* in the selected case. The name of this function is mnemonic for "continuable (exhaustive) type case."

Here is an example:

```
* (setq x 1/3)
1/3
* (ctypecase x
            (integer (* x 4))
            (symbol  (symbol-value x)))

1/3 fell through CTYPECASE expression.
Wanted one of (SYMBOL INTEGER).

Restarts:
  0: Supply a new value for X.
  1: Return to Top-Level.

Debug  (type H for help)
(LISP::CASE-BODY-ERROR CTYPECASE X 1/3 (OR SYMBOL INTEGER)...)
0] restart 0
Type a form to be evaluated:
3.7

3.7 fell through CTYPECASE expression.
Wanted one of (SYMBOL INTEGER).

Restarts:
  0: Supply a new value for X.
  1: Return to Top-Level.

Debug  (type H for help)
(LISP::CASE-BODY-ERROR CTYPECASE X 3.699997 (OR SYMBOL INTEGER)...)
0] restart 0
Type a form to be evaluated:
12
48
*
```

ecase *keyform* {*case*}* [*Macro*]

This control construct is similar to **case**, but no explicit **otherwise** or **t** clause is permitted. Each *case* is of the following form:

({ (*key1 key2* ...) | *key* } *form1 form2* ...)

If no case fires, this signals an error of type **type-error** with a message constructed from the cases. This error cannot be continued. To supply your own error message, use **case** with an **otherwise** clause containing a call to **error**. The name of this function stands for "exhaustive case" or "error-checking case."

Here is an example:

```
* (setq x 1/3)
1/3
* (ecase x
    (alpha (foo))
    (omega (bar))
    ((zeta phi) (baz)))
1/3 fell through ECASE expression.
Wanted one of (ZETA PHI OMEGA ALPHA).

Restarts:
  0: Return to Top-Level.

Debug  (type H for help)
(LISP::%EVAL (ERROR 'CONDITIONS::CASE-FAILURE
                    :NAME 'ECASE :DATUM ...))
0]
```

ccase *keyplace* {*case*}* [*Macro*]

This control construct is similar to **case**, but no explicit **otherwise** or **t** clause is permitted. The *keyplace* must be a generalized variable reference acceptable to **setf**. Each *case* is of the following form:

({ (*key1 key2 ...*) | *key* } *form1 form2 ...*)

If no clause fires, **ccase** signals an error of type **type-error** with a message constructed from the clauses. The user may continue this error using the **store-value** restart.

In this situation **store-value** takes an argument or prompts the user for it and stores the value in *keyplace*, continuing within **ccase** which starts over possibly signalling an error again. Subforms of *keyplace* may evaluate multiple times because of the implicit loop generated.

This returns any values returned by the last *form* in the selected case. The name of this function is mnemonic for "continuable (exhaustive) case."

4.8. Establishing Restarts

with-simple-restart (*name format-string* **&rest** *format-arguments*) {*form*}* [*Macro*]

This macro is shorthand for a common use of **restart-case**. *Name* is the name of the restart, and if this one executes, control returns to **with-simple-restart** returning the values **nil** and **t**. If each *form* executes normally, then the values of the last one are returned.

Name may be **nil**, in which case, this establishes an anonymous restart.

By way of example, you could define **with-simple-restart** in the following way:

```
(defmacro with-simple-restart ((restart-name format-string
                                &rest format-arguments)
                               &body forms)
  `(restart-case (progn ,@forms)
     (,restart-name ()
       :report (lambda (stream)
                 (format stream ,format-string ,@format-arguments))
       (values nil t))))
```

Here is an example of its use:

```
* (defun read-eval-print-loop (level)
    (with-simple-restart (abort "Exit command level ~D." level)
      (loop
        (with-simple-restart (abort "Return to command level ~D."
                                    level)
          (let ((form (prog2 (fresh-line) (read) (fresh-line))))
            (prin1 (eval form)))))))
READ-EVAL-PRINT-LOOP
* (read-eval-print-loop 1)
(+ 'a 3)

Error in function +.
Wrong type argument, A, should have been of type NUMBER.

Restarts:
  0: Return to command level 1.
  1: Exit command level 1.
  2: Return to Top-Level.

Debug  (type H for help)
(CONDITIONS::MAKE-ERROR-TABLE + 0 A NIL)
0] restart 0
(+ 5 nil)

Error in function +.
Wrong type argument, NIL, should have been of type NUMBER.

Restarts:
  0: Return to command level 1.
  1: Exit command level 1.
  2: Return to Top-Level.

Debug  (type H for help)
(CONDITIONS::MAKE-ERROR-TABLE + 0 NIL NIL)
0] restart 1
NIL
T
*
```

restart-case *expression* { (*case-name arglist* {*keyword value*}* {*form*}*) }* [*Macro*]

This macro evaluates *expression* in a dynamic context where the clauses have special meanings as points to which handlers and users may transfer program control. If *expression* executes normally, **restart-case** returns any values returned by it. If anyone invokes one of the restarts, the system transfers control to that branch executing each *form* and returning any values returned by the last such *form*.

If there are no forms in a selected clause, **restart-case** returns **nil**.

Case-name may be **nil** or a symbol naming the restart. A *case-name* may be repeated, in which case **find-restart** will find the first such clause that appears textually. The other clauses are accessible using **compute-restarts**.

Each *arglist* is a normal lambda list of locals to be bound during the execution of its corresponding forms. These arguments convey any necessary data from a call to **invoke-restart** to the clause.

Valid *keyword/value* pairs are as follows:

:interactive exp
By default, **invoke-restart-interactively** passes no arguments to a restart, and all the arguments must be optional to accomodate interactive restarting, what typically occurs in the debugger with user intervention. The arguments may be required if the restart specifies the **:interactive** *keyword*, and **exp** must be a suitable argument to the **function** special form. **Restart-case** evaluates the expression **(function exp)** in the current lexical environment. It should return a function of no arguments that returns a list of values to which **invoke-restart-interactively** will apply the restart. This function runs in the dynamic environment available prior to any restart attempt. The interactive function may use the ***query-io*** stream.

:report exp If **exp** is not a literal string, it must be a suitable argument to the **function** special form, and **restart-case** evaluates the expression **(function exp)** in the current lexical environment. The function takes one required argument, a stream, and the system invokes the function whenever it prints the restart with ***print-escape*** bound to **nil**. If **exp** is a literal string, it is a shorthand for the following:

```
(lambda (stream)
  (write-string exp stream))
```

If the system reports a named restart, and it has no report method, the system uses the restart name in generating default report text. It is an error to define an unnamed restart without any report information since these are generally only useful interactively, possibly as an option for the user in the debugger.

Here are some examples:

```
(loop
  (restart-case (return (apply function some-args))
    (new-function (new-function)
                  :report "Use a different function."
                  :interactive
                  (lambda
                    () (list (prompt-for 'function "Function: ")))
      (setq function new-function))))

(loop
  (restart-case (return (apply function some-args))
    (nil (new-function)
         :report "use a different function."
         :interactive
         (lambda () (list (prompt-for 'function "function: ")))
      (setq function new-function))))

(restart-case (a-command-loop)
  (return-from-command-level
      ()
      :report
      (lambda (stream)
        (format stream "Return from command level ~D." level))
    nil))

(loop
  (restart-case (another-random-computation)
    (continue ()
      nil)))
```

prompt-for is immaterial to this example and does what you think. The first and second examples are equivalent from the point of view of someone using the interactive debugger, but differ in one important

aspect for non-interactive handling; a handler can make use of named restarts other than `nil` as in the following piece of code:

```
(if (find-restart 'new-function)
    (invoke-restart 'new-function the-replacement))
```

This works for the first one, but the second one is only callable interactively, such as from the debugger.

Here is a more complete example:

```
(let ((my-food 'milk)
      (my-color 'greenish-blue))
  (do ()
      ((not (bad-food-color-p my-food my-color)))
    (restart-case (error 'bad-food-color
                         :food my-food :color my-color)
      (use-food  (new-food)
                 :report "Use another food."
                 (setq my-food new-food))
      (use-color (new-color)
                 :report "Use another color."
                 (setq my-color new-color))))
  ;; Can't get here until my-food and my-color are compatible.
  (list my-food my-color))
```

Handlers written for `bad-food-color` errors can use the `find-restart/invoke-restart` idiom to supply a different food for the given color or a different color for the given food. See section 4.2.7 for a discussion of encapsulating this idiom for programmer and user convenience.

`restart-bind` ({*binding*}*) {*form*}* [*Function*]

This macro executes each *form* in a dynamic context where the given restart bindings are in effect. Each binding is of the following form:

(*name function* {*keyword value*}*)

Name may be `nil` to indicate an anonymous restart, or some other symbol to indicate a named restart. *Function* should evaluate to a function that performs the restart. If invoked, this function either transfers control non-locally or simply returns, and it takes whatever arguments the programmer desires. `Invoke-restart` and `invoke-restart-interactively` are the only ways to call it, either from a piece of code or as the result of a debugger command. In the case of interactive invocation, where the arguments are not supplied, the second function named calls the `:interactive-function` option (see below).

The valid *keyword/value* pairs are:

`:interactive-function` *form*
: The *form* evaluates in the current lexical environment and should return a function of no arguments that returns a list of arguments to which `invoke-restart-interactively` applies the restart function. The function may prompt using `*query-io*`.

`:report-function` *form*
: The *form* evaluates in the current lexical environment and should return a function that takes a stream as an argument and prints on it a summary of the action that this restart will take. The system calls this function whenever the restart is printed with `*print-escape*` bound to `nil`.

This is considered a significantly lower-level primitive than `restart-case`, and its intended purpose is that of building higher-level abstractions such as `restart-case`. It still has uses for inclusion in general coding, but typically it appears in macros that define other constructs.

4.9. Finding and Manipulating Restarts

compute-restarts [*Function*]

This function returns a list of the restarts currently active in the dynamic state of a program, see **restart-bind** and **restart-case**. Each restart represents a function that performs some recovery action, typically a dynamic transfer of control. Restart objects are implementation-dependent, but they always have dynamic extent relative to the scope of the binding form.

The result of **compute-restarts** is ordered from more recently established restarts to those first established in time. All elements of the list are valid, including anonymous restarts, even though some may have the same name as others and would not be found by **find-restart** because of this.

Portable programs do not rely on whether multiple calls to **compute-restarts** in the same dynamic environment share elements or are disjoint (not **eq**), and it is an error to modify the resulting list.

restart-name *restart* [*Function*]

This function returns the name of the given **restart** object. If it is unnamed, this returns **nil**.

find-restart *identifier* [*Function*]

This function searches for a particular restart in the current dynamic environment.

If *identifier* is a symbol, this returns the most recently established restart with that name. If none is found, this returns **nil**.

If *identifier* is a **restart** object, this returns the object if it is currently active. If it is inactive, this returns **nil**.

Although anonymous restarts have **nil** as a name, it is an error to supply the symbol **nil** for *identifier*. If your application seems to require this, consider rewriting it to use **compute-restarts**.

invoke-restart *restart* **&rest** *arguments* [*Function*]

This function calls the function associated with *restart* on *arguments*. *Restart* must be a **restart** object or the non-**nil** name of a currently valid restart. If the argument is invalid, this signals a **control-error** error. Note, restart functions (see section 4.2.7), such as **abort** and **continue**, call this function, not vice versa.

invoke-restart-interactively *restart* [*Function*]

This function invokes the function associated with *restart*. If *restart* has an associated interactive method (see **restart-bind** (page 46) and **restart-case** (page 44)), this function calls the method to provide arguments for *restart*'s function. *Restart* must be a **restart** object or the non-**nil** name of a restart that is valid in the current dynamic context. If it is invalid, this function signals an error of type **control-error**.

If no interactive method is associated with *restart*, then it is an error for the *restart*'s function to require arguments.

4.10. Restart Functions

abort [*Function*]

This function transfers control to the restart named **abort**, and if none exists, it signals an error of type **control-error**. This is generally used to return to previous command levels.

continue [*Function*]

This function transfers control to the restart named **continue**, and if none exists, it returns **nil**. This is generally used with simple and *obvious* restarts, such as in **break** and **cerror**, whether in user or system code.

muffle-warning [*Function*]

This function transfers control to the restart named **muffle-warning**, and if none exists, it signals an error of type **control-error**. **Warn** signals **warning** conditions in an environment where this restart causes **warn** to immediately return.

store-value *value* [*Function*]

This function transfers control, passing *value*, to the restart named **store-value**, and if none exists, it returns **nil**. Code that signals errors of type **cell-error** and **type-error** may establish this restart for handlers that can supply replacement data to be stored permanently to correct the situation.

use-value *value* [*Function*]

This function transfers control, passing *value*, to the restart named **use-value**, and if none exists, it returns **nil**. Code that signals **cell-error** errors may establish this restart for handlers that can supply a replacement value to be used once only to correct the situation.

4.11. Debugging Utilities

break &optional *format-string* **&rest** *format-arguments* [*Function*]

This function prints the message described by *format-string* and *format-arguments* and then enters the debugger. While in the debugger, there is a **continue** restart that causes **break** to return **nil** immediately. If *format-string* is unsupplied, this generates a default message.

By way of example, **break** could be defined as follows:

```
(defun break (&optional (format-string "Break")
                        &rest format-arguments)
  (with-simple-restart (continue "Return from BREAK.")
    (invoke-debugger
      (make-condition 'simple-condition
                      :format-string format-string
                      :format-arguments format-arguments)))
    nil)
```

invoke-debugger *condition* [*Function*]
debugger-hook [*Variable*]

This function invokes an interactive mechanism for handling *condition*, which must be a **condition** object. This never directly returns; some non-local transfer of control must occur, such as the use of a restart, aborting to top level, etc.

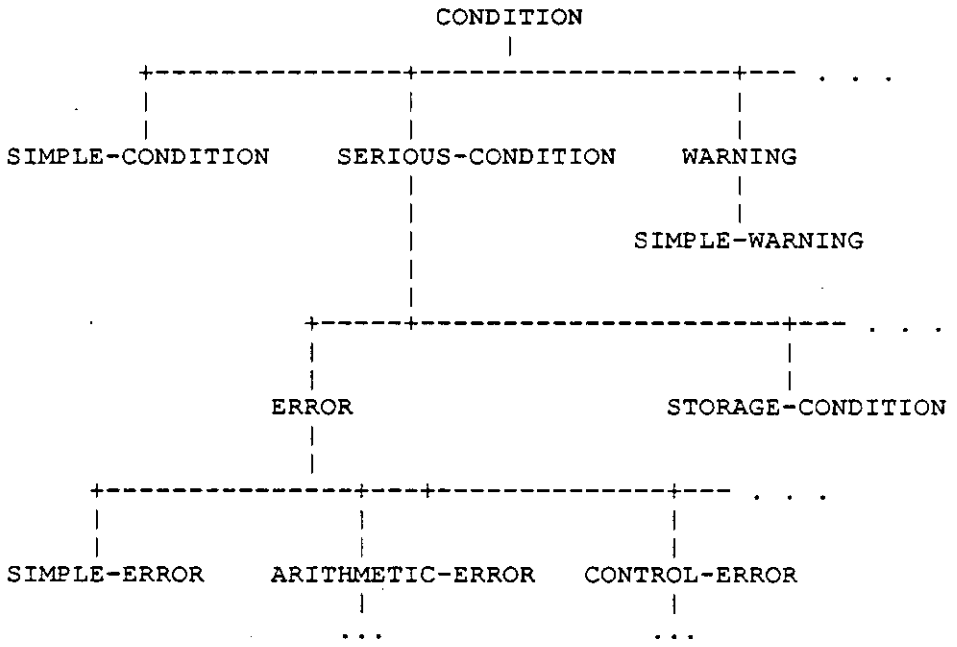
When the variable ***debugger-hook*** is non-**nil**, it is a function **invoke-debugger** calls instead of executing any standard debugger interface. The function takes *condition* and the value of ***debugger-hook*** as arguments, and if it returns, **invoke-debugger** enters the standard debugger anyway. While executing ***debugger-hook***, this variable is **nil**, so if this interface evaluates code

on the user's behalf, it may want to rebind ***debugger-hook*** to the second value passed in to handle recursive errors with the same interface.

4.12. System Defined Types

Restart is the data type used to represent a restart.

A sketch of the **condition** type hierarchy looks like this:



Typically programs do not directly instantiate conditions of the non-terminal types in the above tree (for example **condition**, **warning**, **storage-condition**, **error**, **arithmetic-error**, etc.); the system provides these primarily for type inclusion purposes.

The design of the condition system permits implementations to support non-portable synonyms for these types, as well as to introduce other types above, below, or between the types shown in this tree as long as the indicated subtype relationships are not violated.

The types **simple-condition**, **serious-condition**, and **warning** are pairwise disjoint. The type **error** is disjoint from types **simple-condition** and **warning**.

The following describes all the predefined condition types:

condition All types of conditions, whether **error** or non-**error**, must inherit from this type.

warning All types of warnings inherit from this type. This is a subtype of **condition**.

serious-condition
All serious conditions (conditions serious enough to require interactive intervention if not handled) inherit from this type. This is a subtype of **condition**.

error All types of **error** conditions inherit from this condition. This is a subtype of **serious-condition**.

simple-condition
Conditions signalled by **signal** when given a **format** string as a first argument are of this type. This is a subtype of **condition**. The system supports the initialization keywords

:format-string and :format-arguments for the slots, which can be accessed using simple-condition-format-string and simple-condition-format-arguments. If :format-arguments is unsupplied with make-condition, the format-arguments slot defaults to nil.

simple-warning
Conditions signalled by warn when given a format string as a first argument are of this type. This is a subtype of warning. The system supports the initialization keywords :format-string and :format-arguments for the slots, which can be accessed using simple-condition-format-string and simple-condition-format-arguments. If :format-arguments is unsupplied with make-condition, the format-arguments slot defaults to nil. In implementations supporting multiple inheritance, this type will also be a subtype of simple-condition.

simple-error Conditions signalled by error and cerror when given a format string as a first argument are of this type. This is a subtype of error. The system supports the initialization keywords :format-string and :format-arguments for the slots, which can be accessed using simple-condition-format-string and simple-condition-format-arguments. If :format-arguments is unsupplied with make-condition, the format-arguments slot defaults to nil. In implementations supporting multiple inheritance, this type will also be a subtype of simple-condition.

storage-condition
Conditions related to storage overflow inherit from this type. This is a subtype of serious-condition.

type-error Errors in the transfer of data in a program inherit from this type. This is a subtype of error. For example, conditions signalled by check-type inherit from this type. The system supports the initialization keywords :datum and :expected-type for the slots, which can be accessed using type-error-datum and type-error-expected-type.

simple-type-error
Conditions signalled by facilities similar to check-type may use this type. The system supports the initialization keywords :format-string and :format-arguments for the slots, which can be accessed using simple-condition-format-string and simple-condition-format-arguments. If :format-arguments is unsupplied with make-condition, the format-arguments slot defaults to nil. In implementations supporting multiple inheritance, this type will also be a subtype of simple-condition.

program-error Errors related to incorrect program syntax statically detectable inherit from this type, regardless of whether they are statically detected. This is a subtype of error. This is not a subtype of control-error. The errors resulting from naming a go tag or return-from tag which is not lexically apparent are program errors.

control-error Errors in the dynamic transfer of control in a program inherit from this type. This is a subtype of error. This is not a subtype of program-error. The errors resulting from giving throw a tag which is not active or from giving go or return-from a tag which is no longer dynamically available are control errors.

package-error Errors occurring during operations on packages inherit from this type. This is a subtype of error. The system supports the initialization keyword :package for the slot, which can be accessed using package-error-package.

stream-error Errors occurring during input from, output to, or closing a stream inherit from this type. This is a subtype of error. The system supports the initialization keyword :stream for the slot, which can be accessed using stream-error-stream.

end-of-file The error resulting when reading from a stream with no more input inherits from this type. This is a subtype of stream-error.

file-error Errors occurring during an attempt to open a file, or during some low-level transaction with a file system, inherit from this type. This is a subtype of error. The system supports the initialization keyword :pathname for the slot, which can be accessed using file-error-pathname.

cell-error Errors occurring while accessing a location inherit from this type. This is a subtype of **error**. The system supports the initialization keyword **:name** for the slot, which can be accessed using **cell-error-name**.

unbound-variable
The error resulting from trying to access the value of an unbound variable inherits from this type. This is a subtype of **cell-error**.

undefined-function
The error resulting from trying to access the value of an undefined function inherit from this type. This is a subtype of **cell-error**.

arithmetic-error
Errors occurring while doing arithmetic type operations inherit from this type. This is a subtype of **error**. The system supports the initialization keywords **:operation** and **:operands** for the slots, which can be accessed using **arithmetic-error-operation** and **arithmetic-error-operands**.

division-by-zero
Errors occurring because of division by zero inherit from this type. This is a subtype of **arithmetic-error**.

floating-point-overflow
Errors occurring because of floating point overflow inherit from this type. This is a subtype of **arithmetic-error**.

floating-point-underflow
Errors occurring because of floating point underflow inherit from this type. This is a subtype of **arithmetic-error**.

Chapter 5

The Debugger

By Robert MacLachlan

5.1. Introduction

The CMU Common Lisp debugger is unique in its level of support for source-level debugging of compiled code. Although some other debuggers allow access of variables by name, this seems to be the first Common Lisp debugger that:

- Tells you when a variable doesn't have a value because it hasn't been initialized yet or has already been deallocated, or
- Can display the precise source location corresponding to a code location in the debugged program.

These features allow the debugging of compiled code to be made almost indistinguishable from interpreted code debugging.

The debugger is an interactive command loop that allows a user to examine the function call stack. The debugger is invoked when:[1]

- A `serious-condition` is signalled, and it is not handled, or
- `error` is called, and the condition it signals is not handled, or
- The debugger is explicitly invoked with the COMMON LISP `break` or `debug` functions.

When you enter the debugger, it looks something like this:

```
Error in function CAR.
Wrong type argument, 3, should have been of type LIST.

Restarts:
  0: Return to Top-Level.

Debug  (type H for help)

(CAR 3)
7]
```

The first group of lines describe what the error was that put us in the debugger. In this case `car` was called on `3`. After `Restarts:` is a list of all the ways that we can restart execution after this error. In this case, the only option is to return to top-level. After printing its banner, the debugger prints the current frame and the debugger prompt.

[1]The debugger cannot be used in Hemlock's eval mode, but can be used in slave Lisps running under Hemlock.

5.2. The Command Loop

The debugger is an interactive read-eval-print loop much like the normal top-level, but some symbols are interpreted as debugger commands instead of being evaluated. A debugger command starts with the symbol name of the command, possibly followed by some arguments on the same line. Some commands prompt for additional input.

The debugger prompt is "*frame*] ", where *frame* is the number of the current frame. Frames are numbered starting from zero at the top (most recent call), increasing down to the bottom. The current frame is the frame that commands refer to. The current frame also provides the lexical environment for evaluation of non-command forms.

The package is not significant in debugger commands; any symbol with the name of a debugger command will work. If you want to show the value of a variable that happens also to be the name of a debugger command, you can use the "`L`" command or the `debug:var` function, or you can wrap the variable in a `progn` to hide it from the command loop.

The debugger evaluates forms in the lexical environment of the functions being debugged. The debugger can only access variables. You can't `go` or `return-from` into a function, and you can't call local functions. Special variable references are evaluated with their current value (the innermost binding around the debugger invocation) — you don't get the value that the special had in the current frame. See section 5.4 for more information on debugger variable access.

5.3. Stack Frames

A stack frame is the run-time representation of a call to a function; the frame stores the state that a function needs to remember what it is doing. Frames have:

- Variables (see section 5.4), which are the values being operated on, and
- Arguments to the call (which are really just particularly interesting variables), and
- A current location (see section 5.5), which is the place in the program where the function was running when it stopped to call another function, or because of an interrupt or error.

5.3.1. Stack Motion

These commands move to a new stack frame and print the name of the function and the values of its arguments in the style of a Lisp function call:

U Move up to the next higher frame. More recent function calls are considered to be higher on the stack.

D Move down to the next lower frame.

T Move to the highest frame.

B Move to the lowest frame.

F [*n*]

Move to the frame with the specified number. Prompts for the number if not supplied.

5.3.2. How Arguments are Printed

A frame is printed to look like a function call, but with the actual argument values in the argument positions. So the frame for this call in the source:

```
(myfun (+ 3 4) 'a)
```

would look like this:

```
(MYFUN 7 A)
```

All keyword and optional arguments are displayed with their actual values; if the corresponding argument was not supplied, the value will be the default. So this call:

```
(subseq "foo" 1)
```

would look like this:

```
(SUBSEQ "foo" 1 3)
```

And this call:

```
(string-upcase "test case")
```

would look like this:

```
(STRING-UPCASE "test case" :START 0 :END NIL)
```

The arguments to a function call are displayed by accessing the argument variables. Although those variables are initialized to the actual argument values, they can be set inside the function; in this case the new value will be displayed.

&rest arguments are handled somewhat differently. The value of the rest argument variable is displayed as the spread-out arguments to the call, so:

```
(format t "~A is a ~A." "This" 'test)
```

would look like this:

```
(FORMAT T "~A is a ~A." "This" 'TEST)
```

Rest arguments cause an exception to the normal display of keyword arguments in functions that have both **&rest** and **&key** arguments. In this case, the keyword argument variables are not displayed at all; the rest arg is displayed instead. So for these functions, only the keywords actually supplied will be shown, and the values displayed will be the argument values, not values of the (possibly modified) variables.

If the variable for an argument is never referenced by the function, it will be deleted. The variable value is then unavailable, so the debugger prints **<unused-arg>** instead of the value. Similarly, if for any of a number of reasons (described in more detail in section 5.4) the value of the variable is unavailable or not known to be available, then **<unavailable-arg>** will be printed instead of the argument value.

Printing of argument values is controlled by ***debug-print-level*** and ***debug-print-length*** (page 62).

5.3.3. Function Names

If a function is defined by **defun**, **labels**, or **flet**, then the debugger will print the actual function name after the open parenthesis, like:

```
(STRING-UPCASE "test case" :START 0 :END NIL)
((SETF AREF) #\a "for" 1)
```

Otherwise, the function name is a string, and will be printed in quotes:

```
("DEFUN MYFUN" BAR)
("DEFMACRO DO" (DO ((I 0 (1+ I))) ((= I 13))) NIL)
("SETQ *GC-NOTIFY-BEFORE*")
```

This string name is derived from the **def***mumble* form that encloses or expanded into the lambda, or the outermost enclosing form if there is no **def***mumble*.

5.3.4. Funny Frames

Sometimes the evaluator introduces new functions that are used to implement a user function, but are not directly specified in the source. The main place this is done is for checking argument type and syntax. Usually these functions do their thing and then go away, and thus are not seen on the stack in the debugger. But when you get some sort of error during lambda-list processing, you end up in the debugger on one of these funny frames.

These funny frames are flagged by printing " [*keyword*] " after the parentheses. For example, this call:

```
(car 'a 'b)
```

will look like this:

```
(CAR 2 A) [:EXTERNAL]
```

And this call:

```
(string-upcase "test case" :end)
```

would look like this:

```
("DEFUN STRING-UPCASE" "test case" 335544424 1) [:OPTIONAL]
```

As you can see, these frames have only a vague resemblance to the original call. Fortunately, the error message displayed when you enter the debugger will usually tell you what problem is (in these cases, too many arguments and odd keyword arguments.) Also, if you go down the stack to the frame for the calling function, you can display the original source (see section 5.5.)

With recursive or block compiled functions (see section 7.6.5), an **:EXTERNAL** frame may appear before the frame representing the first call to the recursive function or entry to the compiled block. This is a consequence of the way the compiler does block compilation: there is nothing odd with your program. You will also see **:CLEANUP** frames during the execution of **unwind-protect** cleanup code. Note that inline expansion and open-coding affect what frames are present in the debugger, see sections 5.6 and 6.8.

5.3.5. Tail Recursion

Both the compiler and the interpreter are "properly tail recursive." If a function call is in a tail-recursive position, the stack frame will be deallocated *at the time of the call*, rather than after the call returns. Consider this backtrace:

```
(BAR ...)
(FOO ...)
```

Because of tail recursion, it is not necessarily the case that **FOO** directly called **BAR**. It may be that **FOO** called some other function **FOO2** which then called **BAR** tail-recursively, as in this example:

```
(defun foo ()
  ...
  (foo2 ...)
  ...)

(defun foo2 (...)
  ...
  (bar ...))

(defun bar (...)
  ...)
```

Usually the elimination of tail-recursive frames makes debugging more pleasant, since theses frames are mostly uninformative. If there is any doubt about how one function called another, it can usually be eliminated by finding the source location in the calling frame (section 5.5.)

For a more thorough discussion of tail recursion, see section 7.5.

5.3.6. Unknown Locations and Interrupts

The debugger operates using special debugging information attached to the compiled code. This debug information tells the debugger what it needs to know about the locations in the code where the debugger can be invoked. If the debugger somehow encounters a location not described in the debug information, then it is said to be *unknown*. If the code location for a frame is unknown, then some variables may be inaccessible, and the source location cannot be precisely displayed.

There are three reasons why a code location could be unknown:

- There is inadequate debug information due to the value of the **debug-info** optimization quality. See section 5.6.
- The debugger was entered because of an interrupt such as **^C**.
- A hardware error such as "**bus error**" occurred in code that was compiled unsafely due to the value of the **safety** optimization quality. See section 6.7.1.

In the last two cases, the values of argument variables are accessible, but may be incorrect. See section 5.4.1 for more details on when variable values are accessible.

It is possible for an interrupt to happen when a function call or return is in progress. The debugger may then flame out with some obscure error or insist that the bottom of the stack has been reached, when the real problem is that the current stack frame can't be located. If this happens, return from the interrupt and try again.

5.4. Variable Access

There are three ways to access the current frame's local variables in the debugger. The simplest is to type the variable's name into the debugger's read-eval-print loop. The debugger will evaluate the variable reference as though it had appeared inside that frame.

The debugger doesn't really understand lexical scoping; it has just one namespace for all the variables in a function. If a symbol is the name of multiple variables in the same function, then the reference appears ambiguous, even though lexical scoping specifies which value is visible at any given source location. If the scopes of the two variables are not nested, then the debugger can resolve the ambiguity by observing that only one variable is accessible.

When there are ambiguous variables, the evaluator assigns each one a small integer identifier. The **debug:var** function and the "**L**" command use this identifier to distinguish between ambiguous variables:

L [*prefix*] This command prints the name and value of all variables in the current frame whose name has the specified *prefix*. *prefix* may be a string or a symbol. If no *prefix* is given, then all available variables are printed. If a variable has a potentially ambiguous name, then the name is printed with a "*#identifier*" suffix, where *identifier* is the small integer used to make the name unique.

debug:var *name* **&optional** *identifier* [*Function*]

This function returns the value of the variable in the current frame with the specified *name*. If supplied, *identifier* determines which value to return when there are ambiguous variables.

When *name* is a symbol, it is interpreted as the symbol name of the variable, i.e. the package is significant. If *name* is an uninterned symbol (gensym), then return the value of the uninterned variable with the same name. If *name* is a string, **debug:var** interprets it as the prefix of a variable name, and must unambiguously complete to the name of a valid variable.

This function is useful mainly for accessing the value of uninterned or ambiguous variables, since most variables can be evaluated directly.

5.4.1. Variable Value Availability

The value of a variable may be unavailable to the debugger in portions of the program where COMMON LISP says that the variable is defined. If a variable value is not available, the debugger will not let you read or write that variable. With one exception, the debugger will never display an incorrect value for a variable. Rather than displaying incorrect values, the debugger tells you the value is unavailable.

The one exception is this: if you interrupt (e.g., with `^C`) or if there is an unexpected hardware error such as "`bus error`" (which should only happen in unsafe code), then the values displayed for arguments to the interrupted frame might be incorrect.[2] This exception applies only to the interrupted frame: any frame farther down the stack will be fine.

The value of a variable may be unavailable for these reasons:

- The value of the `debug-info` optimization quality may have omitted debug information needed to determine whether the variable is available. Unless a variable is an argument, its value will only be available when `debug-info` is at least `2`.
- The compiler did lifetime analysis and determined that the value was no longer needed, even though its scope had not been exited. Lifetime analysis is inhibited when the `debug-info` optimization quality is `3`.
- The variable's name is an uninterned symbol (gensym). To save space, the compiler only dumps debug information about uninterned variables when the `debug-info` optimization quality is `3`.
- The frame's location is unknown (see section 5.3.6) because the debugger was entered due to an interrupt or unexpected hardware error. Under these conditions the values of arguments will be available, but might be incorrect. This is the exception above.
- The variable was optimized out of existence. Variables with no reads are always optimized away, even in the interpreter. The degree to which the compiler deletes variables will depend on the value of the `compile-speed` optimization quality, but most source-level optimizations are done under all compilation policies.

Since it is especially useful to be able to get the arguments to a function, argument variables are treated specially when the `speed` optimization quality is less than `3` and the `debug-info` quality is at least `1`. With this compilation policy, the values of argument variables are almost always available everywhere in the function, even at unknown locations. For non-argument variables, `debug-info` must be at least `2` for values to be available, and even then, values are only available at known locations.

5.4.2. Note On Lexical Variable Access

When the debugger command loop establishes variable bindings for available variables, these variable bindings have lexical scope and dynamic extent.[3] You can close over them, but such closures can't be used as upward funargs.

You can also set local variables using `setq`, but if the variable was closed over in the original source and never set, then setting the variable in the debugger may not change the value in all the functions the variable is defined in. Another risk of setting variables is that you may assign a value of a type that the compiler proved the variable could never take on. This may result in bad things happening.

---

[2]Since the location of an interrupt or hardware error will always be an unknown location (see section 5.3.6), non-argument variable values will never be available in the interrupted frame.

[3]The variable bindings are actually created using the COMMON LISP `symbol-macro-let` special form.

5.5. Source Location Printing

One of CMU COMMON LISP's unique capabilities is source level debugging of compiled code. These commands display the source location for the current frame:

source [*context*] This command displays the file that the current frame's function was defined from (if it was defined from a file), and then the source form responsible for generating the code that the current frame was executing. If *context* is specified, then it is an integer specifying the number of enclosing levels of list structure to print.

vsource [*context*] This command is identical to **source**, except that it uses the global values of ***print-level*** and ***print-length*** instead of the debugger printing control variables ***debug-print-level*** and ***debug-print-length***.

The source form for a location in the code is the innermost list present in the original source that encloses the form responsible for generating that code. If the actual source form is not a list, then some enclosing list will be printed. For example, if the source form was a reference to the variable ***some-random-special***, then the innermost enclosing evaluated form will be printed. Here are some possible enclosing forms:

```
(let ((a *some-random-special*))
  ...)

(+ *some-random-special* ...)
```

If the code at a location was generated from the expansion of a macro or a source-level compiler optimization, then the form in the original source that expanded into that code will be printed. Suppose the file **/usr/me/mystuff.lisp** looked like this:

```
(defmacro mymac ()
  '(myfun))

(defun foo ()
  (mymac)
  ...)
```

If **foo** has called **myfun**, and is waiting for it to return, then the **source** would print:

```
File: /usr/me/mystuff.lisp

(MYMAC)
```

Note that the macro use was printed, not the actual function call form, **(myfun)**.

If enclosing source is printed by giving an argument to **source** or **vsource**, then the actual source form is marked by wrapping it in a list whose first element is **#:***HERE*****. In the previous example, **source 1** would print:

```
File: /usr/me/mystuff.lisp

(DEFUN FOO ()
  (#:***HERE***
   (MYMAC))
  ...)
```

5.5.1. How the Source is Found

If the code was defined from Common Lisp by **compile** or **eval**, then the source can always be reliably located. If the code was defined from a **fasl** file created by **compile-file**, then the debugger gets the source forms it prints by reading them from the original source file. This is a potential problem, since the source file might have moved or changed since the time it was compiled.

The source file is opened using the **truename** of the source file pathname originally given to the compiler. This

is an absolute pathname with all logical names and symbolic links expanded. If the file can't be located using this name, then the debugger gives up and signals an error.

If the source file can be found, but has been modified since the time it was compiled, the debugger prints this warning:

```
File has been modified since compilation:
  filename
Using form offset instead of character position.
```

where *filename* is the name of the source file. It then proceeds using a robust but not foolproof heuristic for locating the source. This heuristic works if:

- No top-level forms before the top-level form containing the source have been added or deleted, and
- The top-level form containing the source has not been modified much. (More precisely, none of the list forms beginning before the source form have been added or deleted.)

If the heuristic doesn't work, the displayed source will be wrong, but will probably be near the actual source. If the "shape" of the top-level form in the source file is too different from the original form, then an error will be signalled. When the heuristic is used, the the source location commands are noticeably slowed.

Source location printing can also be confused if (after the source was compiled) a read-macro you used in the code was redefined to expand into something different, or if a read-macro ever returns the same **eq** list twice. If you don't define read macros and don't use ## in perverted ways, you don't need to worry about this.

5.5.2. Source Location Availability

Source location information is only available when the **debug-info** optimization quality is at least **2**. If source location information is unavailable, the source commands will give an error message.

If source location information is available, but the source location is unknown because of an interrupt or unexpected hardware error (see section 5.3.6), then the command will print:

```
Unknown location: using block start.
```

and then proceed to print the source location for the start of the *basic block* enclosing the code location. It's a bit complicated to explain exactly what a basic block is, but here are some properties of the block start location:

- The block start location may be the same as the true location.
- The block start location will never be later in the the program's flow of control than the true location.
- No conditional control structures (such as **if**, **cond**, **or**) will intervene between the block start and the true location (but note that some conditionals present in the original source could be optimized away.) Function calls *do not* end basic blocks.
- The head of a loop will be the start of a block.
- The programming language concept of "block structure" and the COMMON LISP **block** special form are totally unrelated to the compiler's basic block.

In other words, the true location lies between the printed location and the next conditional (but watch out because the compiler may have changed the program on you.)

5.6. Compiler Policy Control

The compilation policy specified by **optimize** declarations affects the behavior seen in the debugger. The **debug-info** quality directly affects the debugger by controlling the amount of debugger information dumped. Other optimization qualities have indirect but observable effects due to changes in the way compilation is done.

Unlike the other optimization qualities (which are compared in relative value to evaluate tradeoffs), the

debug-info optimization quality is directly translated to a level of debug information. This absolute interpretation allows the user to count on a particular amount of debug information being available even when the values of the other qualities are changed during compilation. These are the levels of debug information that correspond to the values of the **debug-info** quality:

0	Only the function name and enough information to allow the stack to be parsed.
1	Level 0 plus all argument variables. Values will only be accessible if the argument variable is never set and **speed** is not 3.
2	Level 1 plus all interned local variables, source location information, and lifetime information that tells the debugger when arguments are available (even when **speed** is 3 or the argument is set.) This is the default.
3	Level 2 plus all uninterned variables. In addition, lifetime analysis is disabled (even when **speed** is 3), ensuring that all variable values are available at any known location within the scope of the binding. This has a speed penalty in addition to the obvious space penalty.

As you can see, if the **speed** quality is 3, debugger performance is degraded. This effect comes from the elimination of argument variable special-casing (see section 5.4.1.) Some degree of speed/debuggability tradeoff is unavoidable, but the effect is not too drastic when **debug-info** is at least 2.

In addition to **inline** and **notinline** declarations, the relative values of the **speed** and **space** qualities also change whether functions are inline expanded (see section 7.7.) If a function is inline expanded, then there will be no frame to represent the call, and the arguments will be treated like any other local variable. Functions may also be "semi-inline", in which case there is a frame to represent the call, but the call is to an optimized local version of the function, not to the original function.

5.7. Exiting Commands

These commands get you out of the debugger.

q	Throw to top level.
restart [*n*]	Invokes the *n*th restart case as displayed by the **error** command. If *n* is not specified, the available restart cases are reported.
go	Calls **continue** on the condition given to **debug**. If there is no restart case named *continue*, then an error is signaled.
abort	Effectively calls **abort** on the condition given to **debug**. This is useful for popping debug command loop levels or aborting to top level, as the case may be.

5.8. Information Commands

Most of these commands print information about the current frame or function, but a few show general information.

h	Displays a synopsis of debugger commands.
p	Displays the current function call as it would be displayed by moving to this frame.
pp	Displays the current function call using ***print-level*** and ***print-length*** instead of ***debug-print-level*** and ***debug-print-length***.
error	Prints the condition given to **invoke-debugger** and the active proceed cases.
backtrace [*n*]	Displays all the frames from the current to the bottom. Only shows *n* frames if specified. The printing is controlled by ***debug-print-level*** and ***debug-print-length***.

5.9. Specials

These are the special variables that control the debugger action.

extensions:*debug-print-level* [*Variable*]
extensions:*debug-print-length* [*Variable*]

print-level and ***print-length*** are bound to these values during the execution of some debug commands. When evaluating arbitrary expressions in the debugger, the normal values of ***print-level*** and ***print-length*** are in effect. These variables are initially set to 3 and 5, respectively.

5.10. Function Tracing

The tracer causes selected functions to print their arguments and their results whenever they are called. Options allow conditional printing of the trace information and conditional breakpoints on function entry. Currently, tracing of **setf** functions is not supported.

trace &rest *specs* [*Macro*]

Invokes tracing on the specified functions, and pushes their names onto the global list in ***traced-function-list***. Each *spec* is either the name of a function, or the form

```
(function-name
   trace-option-name value
   trace-option-name value
   ...)
```

If no *specs* are given, **trace** returns the list of all currently traced functions, ***traced-function-list***.

If a function is traced with no options, then each time it is called, a single line containing the name of the function, the arguments to the call, and the depth of the call will be printed on the stream ***trace-output***. After it returns, another line will be printed which contains the depth of the call and all of the return values. The lines are indented to highlight the depth of the calls.

Trace options can cause the normal printout to be suppressed, or cause extra information to be printed. Each traced function carries its own set of options which is independent of the options given for any other function. Every time a function is specified in a call to trace, all of the old options are discarded. The available options are:

:condition A form to eval before before each call to the function. Trace printout will be suppressed whenever the form returns **nil**.

:break A form to eval before each call to the function. If the form returns non **nil**, then a breakpoint loop will be entered immediately before the function call.

:break-after Like **:break**, but the form is evaled and the break loop invoked after the function call.

:break-all A form which should be used as both the **:break** and the **:break-after** args.

:wherein A function name or a list of function names. Trace printout for the traced function will only occur when it is called from within a call to one of the **:wherein** functions.

:print A list of forms which will be evaluated and printed whenever the function is called. The values are printed one per line, and indented to match the other trace output. This printout will be suppressed whenever the normal trace printout is suppressed.

`:print-after` Like `:print` except that the values of the forms are printed whenever the function exits.

`:print-all` This is used as the combination of `:print` and `:print-after`.

`untrace &rest` *function-names* [*Macro*]
This macro turns off tracing for the specified functions, and removes their names from `*traced-function-list*`. If no *function-names* are given, then all currently traced functions are untraced.

`extensions:*traced-function-list*` [*Variable*]
A list of function names maintained and used by `trace`, `untrace`, and `untrace-all`. This list should contain the names of all functions currently being traced.

`extensions:*trace-print-level*` [*Variable*]
`extensions:*trace-print-length*` [*Variable*]
These variables control the values of `*print-level*` and `*print-length*` when printing trace output. The forms printed by the `:print` options are also affected. `*trace-print-level*` and `*trace-print-length*` are initially set to `nil`. When null, the global values of `*print-level*` and `*print-length*` are used.

`extensions:*max-trace-indentation*` [*Variable*]
The maximum number of spaces which should be used to indent trace printout. This variable is initially set to 40.

5.10.1. Encapsulation Functions

The encapsulation functions provide a mechanism for intercepting the arguments and results of a function. `encapsulate` changes the function definition of a symbol, and saves it so that it can be restored later. The new definition normally calls the original definition. The COMMON LISP `fdefinition` function always returns the original definition, stripping off any encapsulation. Currently, encapsulation of `setf` functions is not supported.

The original definition of the symbol can be restored at any time by the `unencapsulate` function. `encapsulate` and `unencapsulate` allow a symbol to be multiply encapsulated in such a way that different encapsulations can be completely transparent to each other.

Each encapsulation has a type which may be an arbitrary lisp object. If a symbol has several encapsulations of different types, then any one of them can be removed without affecting more recent ones. A symbol may have more than one encapsulation of the same type, but only the most recent one can be undone.

`extensions:encapsulate` *symbol type body* [*Function*]
Saves the current definition of *symbol*, and replaces it with a function which returns the result of evaluating the form, *body*. *Type* is an arbitrary lisp object which is the type of encapsulation.

When the new function is called, the following variables are bound for the evaluation of *body*:

`extensions:argument-list`
A list of the arguments to the function.

`extensions:basic-definition`
The unencapsulated definition of the function.

The unencapsulated definition may be called with the original arguments by including the form

```
(apply extensions:basic-definition extensions:argument-list)
```

encapsulate always returns *symbol*.

extensions:unencapsulate *symbol type* [*Function*]
Undoes *symbol*'s most recent encapsulation of type *type*. *Type* is compared with **eq**. Encapsulations of other types are left in place.

extensions:encapsulated-p *symbol type* [*Function*]
Returns **t** if *symbol* has an encapsulation of type *type*. Returns **nil** otherwise. *Type* is compared with **eq**.

Chapter 6

The Compiler

6.1. Introduction

This chapter contains information about the compiler that every CMU Common Lisp user should be familiar with. Chapter 7 goes into greater depth, describing ways to use more advanced features.

The CMU Common Lisp compiler (also known as Python) has many features that are seldom or never supported by conventional Common Lisp compilers:

- Source level debugging of compiled code (see chapter 5.)
- Type error compiler warnings for type errors detectable at compile time.
- Compiler error messages that provide a good indication of where the error appeared in the source.
- Full run-time checking of all potential type errors, with optimization of type checks to minimize the cost.
- Scheme-like features such as proper tail recursion and extensive source-level optimization.
- Advanced tuning and optimization features such as comprehensive efficiency notes, flow analysis, and untagged number representations (see chapter 7.)

6.2. Calling the Compiler

Functions may be compiled using **compile**, **compile-file**, or **compile-from-stream**.

compile *name* **&optional** *definition* [*Function*]

This function compiles the function whose name is *name*. If *name* is **nil**, the compiled function object is returned. If *definition* is supplied, it should be a lambda expression that is to be compiled and then placed in the function cell of *name*. As per the proposed X3J13 cleanup "compile-argument-problems", *definition* may also be an interpreted function.

The return values are as per the proposed X3J13 cleanup "compiler-diagnostics". The first value is the function name or function object. The second value is **nil** if no compiler diagnostics were issued, and **t** otherwise. The third value is **nil** if no compiler diagnostics other than style warnings were issued. A non-**nil** value indicates that there were "serious" compiler diagnostics issued, or that other conditions of type **error** or **warning** (but not **style-warning**) were signalled during compilation.

compile-file *input-pathname* **&key :output-file :error-file** [*Function*]
:trace-file :error-output :load
:block-compile

The CMU Common Lisp **compile-file** is extended through the addition of several new keywords and an additional interpretation of *input-pathname*:

input-pathname If this argument is a list of input files, rather than a single input pathname, then all the source files are compiled into a single object file. In this case, the name of the first file is used to determine the default output file names. This is especially useful in combination with *block-compile*.

output-file This argument specifies the name of the output file. **t** gives the default name, **nil** suppresses the output file.

error-file A listing of all the error output is directed to this file. If there are no errors, then no error file is produced (and any existing error file is deleted.) **t** gives "*name*.**err**" (the default), and **nil** suppresses the output file.

error-output If **t** (the default), then error output is sent to ***error-output***. If a stream, then output is sent to that stream instead. If **nil**, then error output is suppressed. Note that this error output is in addition to (but the same as) the output placed in the *error-file*.

trace-file If true, several of the intermediate representations (including annotated assembly code) are dumped out to this file. **t** gives "*name*.**trace**". Trace output is off by default. See section 7.10.5.

load If true, load the resulting output file.

block-compile If true, then the file will be block compiled, resolving function references at compile time. See section 7.6.5.

The return values are as per the proposed X3J13 cleanup "compiler-diagnostics". The first value from **compile-file** is the truename of the output file, or **nil** if the file could not be created. The interpretation of the second and third values is described above for **compile**.

extensions:compile-from-stream input-stream &key :error-stream [*Function*]
:trace-stream
:block-compile

This function is similar to **compile-file**, but it takes all its arguments as streams. It reads Common Lisp code from *input-stream* until end of file is reached, compiling into the current environment. This function returns the same two values as the last two values of **compile**. No output files are produced.

6.3. Compilation Units

CMU Common Lisp supports the **with-compilation-unit** macro added to the language by the proposed X3J13 "with-compilation-unit" compiler cleanup. This provides a mechanism for eliminating spurious undefined warnings when there are forward references across files.

with-compilation-unit ({*key value*}*) {*form*}* [*Macro*]

This macro evaluates the *forms* in an environment that causes warnings for undefined variables, functions and types to be delayed until all the forms have been evaluated. Only one keyword option is recognized, which is **:force**. Any unrecognized options will result in a warning.

If uses of **with-compilation-unit** are dynamically nested, the outermost use will take precedence, effectively causing the inner uses to be equivalent to **progn**. However, when the **force** option is true this shadowing is inhibited; an inner use will print summary warnings for the compilations within the

inner scope.

Warnings about undefined variables, functions and types are delayed until the end of the current compilation unit. The compiler entry functions (`compile`, etc.) implicitly use `with-compilation-unit`, so undefined warnings will be printed at the end of the compilation unless there is an enclosing `with-compilation-unit`. In order the gain the benefit of this mechanism, you should wrap a single `with-compilation-unit` around the calls to `compile-file`, i.e.:

```
(with-compilation-unit ()
  (compile-file "file1")
  (compile-file "file2")
  ...)
```

Unlike for functions and types, undefined warnings for variables are not suppressed when a definition (e.g. `defvar`) appears after the reference (but in the same compilation unit.) This is because doing special declarations out of order just doesn't work — although early references will be compiled as special, bindings will be done lexically.

Undefined warnings are printed with full source context (see section 6.4), which tremendously simplifies the problem of finding undefined references that resulted from macroexpansion. After printing detailed information about the undefined uses of each name, `with-compilation-unit` also prints summary listings of the names of all the undefined functions, types and variables.

`*undefined-warning-limit*` [*Variable*]

This variable controls the number of undefined warnings for each distinct name that are printed with full source context when the compilation unit ends. If there are more undefined references than this, then they are condensed into a single warning:

Warning: *count* **more uses of undefined function** *name*.

When the value is `0`, then the undefined warnings are not broken down by name at all: only the summary listing of undefined names is printed.

6.4. Interpreting Error Messages

One of Python's unique features is the level of source location information it provides in error messages. The error messages contain a lot of detail in a terse format, to they may be confusing at first. Error messages will be illustrated using this example program:

```
(defmacro zoq (x)
  `(roq (ploq (+ ,x 3))))

(defun foo (y)
  (declare (symbol y))
  (zoq y))
```

The main problem with this program is that it is trying to add `3` to a symbol. Note also that the functions `roq` and `ploq` aren't defined anywhere.

6.4.1. The Parts of the Error Message

The compiler will produce this warning:

```
File: /usr/me/stuff.lisp

In: DEFUN FOO
  (ZOQ Y)
--> ROQ PLOQ +
==>
  Y
Warning: Result is a SYMBOL, not a NUMBER.
```

In this example we see each of the six possible parts of a compiler error message:

File: /usr/me/stuff.lisp
: This is the *file* that the compiler read the relevant code from. The file name is displayed because it may not be immediately obvious when there is an error during compilation of a large system, especially when **with-compilation-unit** is used to delay undefined warnings.

In: DEFUN FOO
: This is the *definition* or top-level form responsible for the error. It is obtained by taking the first two elements of the enclosing form whose first element is a symbol beginning with "**DEF**". If there is no enclosing *def*mumble, then the outermost form is used. If there are multiple *def*mumbles, then they are all printed from the out in, separated by =>'s. In this example, the problem was in the **defun** for **foo**.

(ZOQ Y)
: This is the *original source* form responsible for the error. Original source means that the form directly appeared in the original input to the compiler, i.e. in the lambda passed to **compile** or the top-level form read from the source file. In this example, the expansion of the **zoq** macro was responsible for the error.

--> ROQ PLOQ +
: This is the *processing path* that the compiler used to produce the errorful code. The processing path is a representation of the evaluated forms enclosing the actual source that the compiler encoutered when processing the original source. The path is the first element of each form, or the form itself if the form is not a list. These forms result from the expansion of macros or source-to-source transformation done by the compiler. In this example, the enclosing evaluated forms are the calls to **roq**, **ploq** and **+**. These calls resulted from the expansion of the **zoq** macro.

==> Y
: This is the *actual source* responsible for the error. If the actual source appears in the explanation, then we print the next enclosing evaluated form, instead of printing the actual source twice. (This is the form that would otherwise have been the last form of the processing path.) In this example, the problem is with the evaluation of the reference to the variable **y**.

Warning: Result is a SYMBOL, not a NUMBER.
: This is the *explanation* the problem. In this example, the problem is that **y** evaluates to a **symbol**, but is in a context where a number is required (the argument to +).

Note that each part of the error message is distinctively marked:

- **File:** and **In:** mark the file and definition, respectively.
- The original source is an indented form with no prefix.
- Each line of the processing path is prefixed with **-->**.
- The actual source form is indented like the original source, but is marked by a preceding ==> line. This is like the "macroexpands to" notation used in *Common Lisp: The Language*.
- The explanation is prefixed with the error severity (see section 6.4.4), either **Error:**, **Warning:**, or **Note:**.

Each part of the error message is more specific than the preceding one. If consecutive error messages are for nearby locations, then the front part of the error messages would be the same. In this case, the compiler omits as much of the second message as in common with the first. For example:

```
File: /usr/me/stuff.lisp

In: DEFUN FOO
  (ZOQ Y)
--> ROQ
==>
  (PLOQ (+ Y 3))
Warning: Undefined function: PLOQ

==>
  (ROQ (PLOQ (+ Y 3)))
Warning: Undefined function: ROQ
```

In this example, the file, definition and original source are identical for the two messages, so the compiler omits them in the second message. If consecutive messages are entirely identical, then the compiler prints only the first message, followed by:

[Last message occurs *repeats* **times]**

where *repeats* is the number of times the message was given.

If the source was not from a file, then no file line is printed. If the actual source is the same as the original source, then the processing path and actual source will be omitted. If no forms intervene between the original source and the actual source, then the processing path will also be omitted.

6.4.2. The Original and Actual Source

The *original source* displayed will almost always be a list. If the actual source for an error message is a symbol, the original source will be the immediately enclosing evaluated list form. So even if the offending symbol does appear in the original source, the compiler will print the enclosing list and then print the symbol as the actual source (as though the symbol were introduced by a macro.)

When the *actual source* is displayed (and is not a symbol), it will always be code that resulted from the expansion of a macro or a source-to-source compiler optimization. This is code that did not appear in the original source program; it was introduced by the compiler.

Keep in mind that when the compiler displays a source form in an error message, it always displays the most specific (innermost) responsible form. For example, compiling this function:

```
(defun bar (x)
  (let (a)
    (declare (fixnum a))
    (setq a (foo x))
    a))
```

Gives this error message:

```
In: DEFUN BAR
  (LET (A) (DECLARE (FIXNUM A)) (SETQ A (FOO X)) A)
Warning: The binding of A is not a FIXNUM:
  NIL
```

This error message is not saying "there's a problem somewhere in this **let**" — it is saying that there is a problem with the **let** itself. In this example, the problem is that **a**'s **nil** initial value is not a **fixnum**.

6.4.3. The Processing Path

The processing path is mainly useful for debugging macros, so if you don't write macros, you can ignore the processing path. Consider this example:

```
(defun foo (n)
  (dotimes (i n *undefined*)))
```

Compiling results in this error message:

```
In: DEFUN FOO
  (DOTIMES (I N *UNDEFINED*))
--> DO BLOCK LET TAGBODY RETURN-FROM
==>
  (PROGN *UNDEFINED*)
Warning: Undefined variable: *UNDEFINED*
```

Note that `do` appears in the processing path. This is because `dotimes` expands into:

```
(do ((i 0 (1+ i)) (#:g1 n))
    ((>= i #:g1) *undefined*)
  (declare (type unsigned-byte i)))
```

The rest of the processing path results from the expansion of `do`:

```
(block nil
  (let ((i 0) (#:g1 n))
    (declare (type unsigned-byte i))
    (tagbody (go #:g3)
     #:g2    (psetq i (1+ i))
     #:g3    (unless (>= i #:g1) (go #:g2))
             (return-from nil (progn *undefined*)))))
```

In this example, the compiler descended into the `block`, `let`, `tagbody` and `return-from` to reach the `progn` printed as the actual source. This is a place where the "actual source appears in explanation" rule was applied. The innermost actual source form was the symbol `*undefined*` itself, but that also appeared in the explanation, so the compiler backed out one level.

6.4.4. Error Severity

There are three levels of compiler error severity:

Error — This severity is used when the compiler encounters a problem serious enough to prevent normal processing of a form. Instead of compiling the form, the compiler compiles a call to `error`. Errors are used mainly for signalling syntax errors. If an error happens during macroexpansion, the compiler will handle it. The compiler also handles and attempts to proceed from read errors.

Warning — Warnings are used when the compiler can prove that something bad will happen if a portion of the program is executed, but the compiler can proceed by compiling code that signals an error at runtime if the problem has not been fixed:

- Violation of type declarations, or
- Function calls that have the wrong number of arguments or malformed keyword argument lists, or
- Referencing a variable declared `ignore`, or unrecognized declaration specifiers.

In the language of the COMMON LISP standard, these are situations where the compiler can determine that a situation with undefined consequences or that would cause an error to be signalled would result at runtime.

Note — Notes are used when there is something that seems a bit odd, but that might reasonably appear in correct programs.

Note that the compiler does not fully conform to the proposed X3J13 "compiler-diagnostics" cleanup. Errors, warnings and notes mostly correspond to errors, warnings and style-warnings, but many things that the cleanup considers to be style-warnings are printed as warnings rather than notes. Also, warnings, style-warnings and most errors aren't really signalled using the condition system.

6.4.5. Errors During Macroexpansion

The compiler handles errors that happen during macroexpansion, turning them into compiler errors. If you want to debug the error (to debug a macro), you can set ***break-on-signals*** to **error**. For example, this definition:

```
(defun foo (e l)
  (do ((current l (cdr current))
       ((atom current) nil))
      (when (eq (car current) e) (return current))))
```

gives this error:

```
In: DEFUN FOO
  (DO ((CURRENT L #) (# NIL)) (WHEN (EQ # E) (RETURN CURRENT)) )
Error: (during macroexpansion)

Error in function LISP::DO-DO-BODY.
DO step variable is not a symbol: (ATOM CURRENT)
```

6.4.6. Read Errors

The compiler also handles errors while reading the source. For example:

```
Error: Read error at 2:
 "(,/\foo)"
Error in function LISP::COMMA-MACRO.
Comma not inside a backquote.
```

The "**at 2**" refers to the character position in the source file at which the error was signalled, which is generally immediately after the erroneous text. The next line, "**(,/\foo)**", is the line in the source that contains the error file position. The "**/**" indicates the error position within that line (in this example, immediately after the offending comma.)

When in Hemlock (or any other EMACS-like editor), you can go to a character position with "**M-< C-u** *position* **C-f**". Note that if the source is from a Hemlock buffer, then the position is relative to the start of the compiled region or **defun**, not the file or buffer start.

After printing a read error message, the compiler attempts to recover from the error by backing up to the start of the enclosing top-level form and reading again with ***read-suppress*** true. If the compiler can recover from the error, then it substitutes a call to **error** for the unreadable form and proceeds to compile the rest of the file normally.

If there is a read error when the file position is at the end of the file (i.e., an unexpected EOF error), then the error message looks like this:

```
Error: Read error in form starting at 14:
 "(defun test ()"
Error in function LISP::FLUSH-WHITESPACE.
EOF while reading #<Stream for file "/usr/me/test.lisp">
```

In this case, "**starting at 14**" indicates the character position at which the compiler started reading, i.e. the position before the start of the form that was missing the closing delimiter. The line "**(defun test ()**" is first line after the starting position that the compiler thinks might contain the unmatched open delimiter.

6.4.7. Error Message Variables

These variables control the verbosity of error messages. See also ***undefined-warning-limit*** (page 67), ***efficiency-note-limit*** and ***efficiency-note-cost-threshold*** (page 117).

enclosing-source-cutoff [*Variable*]

This variable specifies the number of enclosing actual source forms that are printed in full, rather than in the abbreviated processing path format. Increasing the value from its default of 1 allows you to see more of the guts of the macroexpanded source, which is useful when debugging macros.

error-print-length [*Variable*]

error-print-level [*Variable*]

These variables are the print level and print length used in printing error messages. The default values are 5 and 3. If null, the global values of ***print-level*** and ***print-length*** are used.

source-context-take-car-forms [*Variable*]

When printing the definition part of an error message, this variable is used to determine whether to take the **car** of the first subform if it is a list. The value is a list of the form names for which taking the **car** is appropriate, initially **defstruct** and **function**.

6.5. Types in Python

A big difference between Python and all other Common Lisp compilers is the approach to type checking and amount of knowledge about types:

- Python treats type declarations much differently that other Lisp compilers do. Python doesn't blindly believe type declarations; it considers them assertions about the program that should be checked.
- Python also has a tremendously greater knowledge of the COMMON LISP type system than other compilers. Support is incomplete only for the **not**, **and** and **satisfies** types.

See also sections 7.2 and 7.3.

6.5.1. Compile Time Type Errors

If the compiler can prove at compile time that some portion of the program cannot be executed without a type error, then it will give a warning at compile time. It is possible that the offending code would never actually be executed at run-time due to some higher level consistency constraint unknown to the compiler, so a type warning doesn't always indicate an incorrect program. For example, consider this code fragment:

```
(defun raz (foo)
  (let ((x (case foo
             (:this 13)
             (:that 9)
             (:the-other 42))))
    (declare (fixnum x))
    (foo x)))
```

Compilation produces this warning:

```
In: DEFUN RAZ
  (CASE FOO (:THIS 13) (:THAT 9) (:THE-OTHER 42))
--> LET COND IF COND IF COND IF
==>
  (COND)
Warning: This is not a FIXNUM:
  NIL
```

In this case, the warning is telling you that if **foo** isn't any of **:this**, **:that** or **:the-other**, then **x** will be initialized to **nil**, which the **fixnum** declaration makes illegal. The warning will go away if **ecase** is used instead of **case**, or if **:the-other** is changed to **t**.

This sort of spurious type warning happens moderately often in the expansion of complex macros and in inline

functions. In such cases, there may be dead code that is impossible to correctly execute. The compiler can't always prove this code is dead (could never be executed), so it compiles the erroneous code (which will always signal an error if it is executed) and gives a warning.

extensions:required-argument [*Function*]

This function can be used as the default value for keyword arguments that must always be supplied. Since it is known by the compiler to never return, it will avoid any compile-time type warnings that would result from a default value inconsistent with the declared type. When this function is called, it signals an error indicating that a required keyword argument was not supplied. This function is also useful for **defstruct** slot defaults corresponding to required arguments. See also section 7.2.5.

Although this function is a CMU extension, it is relatively harmless to use it in otherwise portable code, since you can easily define it yourself:

```
(defun required-argument ()
  (error "A required keyword argument was not supplied."))
```

Type warnings are inhibited when the **extensions:inhibit-warnings** optimization quality is 3 (see section 6.7.) This can be used in a local declaration to inhibit type warnings in a code fragment that has spurious warnings.

6.5.2. Precise Type Checking

With the default compilation policy, all type assertions[4] are precisely checked. Precise checking means that the check is done as though **typep** had been called with the exact type specifier that appeared in the declaration. In Python uses *policy* to determine whether to trust type assertions (see section 6.7). Type assertions from declarations are indistinguishable from the type assertions on arguments to built-in functions. In Python, adding type declarations makes code safer.

If a variable is declared to be **(integer 3 17)**, then its value must always always be an integer between 3 and 17. If multiple type declarations apply to a single variable, then all the declarations must be correct; it is as though all the types were intersected producing a single **and** type specifier.

Argument type declarations are automatically enforced. If you declare the type of a function argument, a type check will be done when that function is called. In a function call, the called function does the argument type checking, which means that a more restrictive type assertion in the calling function (e.g., from **the**) may be lost.

The types of structure slots are also checked. The value of a structure slot must always be of the type indicated in any **:type** slot option.[5] Because of precise type checking, the arguments to slot accessors are checked to be the correct type of structure.

In traditional Common Lisp compilers, not all type assertions are checked, and type checks are not precise. Traditional compilers blindly trust explicit type declarations, but may check the argument type assertions for built-in functions. Type checking is not precise, since the argument type checks will be for the most general type legal for that argument. In many systems, type declarations suppress what little type checking is being done, so adding type declarations makes code unsafe. This is a problem since it discourages writing type declarations during initial coding. In addition to being more error prone, adding type declarations during tuning also loses all the benefits of

---

[4]There are a few circumstances where a type declaration is discarded rather than being used as type assertion. This doesn't affect safety much, since such discarded declarations are also not believed to be true by the compiler.

[5]The initial value need not be of this type as long as the corresponding argument to the constructor is always supplied, but this will cause a compile-time type warning unless **required-argument** is used.

debugging with checked type assertions.

To gain maximum benefit from Python's type checking, you should always declare the types of function arguments and structure slots as precisely as possible. This often involves the use of **or**, **member** and other list-style type specifiers. Paradoxically, even though adding type declarations introduces type checks, it usually reduces the overall amount of type checking. This is especially true for structure slot type declarations.

Python uses the **safety** optimization quality (rather than presence or absence of declarations) to choose one of three levels of run-time type error checking: see section 6.7.1. See also section 7.2 for more information about types in Python.

6.5.3. Weakened Type Checking

When the value for the **speed** optimization quality is greater than **safety**, and **safety** is not 0, then type checking is weakened to reduce the speed and space penalty. In structure-intensive code this can double the speed, yet still catch most type errors. Weakened type checks provide a level of safety similar to that of "safe" code in other Common Lisp compilers.

A type check is weakened by changing the check to be for some convenient supertype of the asserted type. For example, **(integer 3 17)** is changed to **fixnum**, **(simple-vector 17)** to **simple-vector**, and structure types are changed to **structure**. A complex check like:

```
(or node hunk (member :foo :bar :baz))
```

will be omitted entirely (i.e., the check is weakened to *.) If a precise check can be done for no extra cost, then no weakening is done.

Although weakened type checking is similar to type checking done by other compilers, it is sometimes safer and sometimes less safe. Weakened checks are done in the same places is precise checks, so all the preceding discussion about where checking is done still applies. Weakened checking is sometimes somewhat unsafe because although the check is weakened, the precise type is still input into type inference. In some contexts this will result in type inferences not justified by the weakened check, and hence deletion of some type checks that would be done by conventional compilers.

For example, if this code was compiled with weakened checks:

```
(defstruct foo
  (a nil :type simple-string))

(defstruct bar
  (a nil :type single-float))

(defun myfun (x)
  (declare (type bar x))
  (* (bar-a x) 3.0))
```

and **myfun** was passed a **foo**, then no type error would be signalled, and we would try to multiply a **simple-vector** as though it were a float (with unpredictable results.) This is because the check for **bar** was weakened to **structure**, yet when compiling the call to **bar-a**, the compiler thinks it knows it has a **bar**.

Note that normally even weakened type checks report the precise type in error messages. For example, if **myfun**'s **bar** check is weakened to **structure**, and the argument is **nil**, then the error will be:

```
Type-error in MYFUN:
  NIL is not of type BAR
```

However, there is some speed and space cost for signalling a precise error, so the weakened type is reported if the **speed** optimization quality is 3 or **debug-info** quality is less than 1:

```
Type-error in MYFUN:
  NIL is not of type STRUCTURE
```

See section 6.7.1 for further discussion of the `optimize` declaration.

6.6. Getting Existing Programs to Run

Since Python does much more comprehensive type checking than other Lisp compilers, Python will detect type errors in many programs that have been debugged using other compilers. These errors are mostly incorrect declarations, although compile-time type errors can find actual bugs if parts of the program have never been tested.

Some incorrect declarations can only be detected by run-time type checking. It is very important to initially compile programs with full type checks and then test this version. After the checking version has been tested, then you can consider weakened type checks or no checking. **This applies even to previously debugged programs.** Python does much more type inference than other Common Lisp compilers, so believing an incorrect declaration does much more damage.

The most common problem is with variables whose initial value doesn't match the type declaration. Incorrect initial values will always be flagged by a compile-time type error, and they are simple to fix once located. Consider this code fragment:

```
(prog (foo)
  (declare (fixnum foo))
  (setq foo ...)
  ...)
```

Here the variable `foo` is given an initial value of `nil`, but is declared to be a `fixnum`. Even if it is never read, the initial value of a variable must match the declared type. There are two ways to fix this problem. Change the declaration:

```
(prog (foo)
  (declare (type (or fixnum null) foo))
  (setq foo ...)
  ...)
```

or change the initial value:

```
(prog ((foo 0))
  (declare (fixnum foo))
  (setq foo ...)
  ...)
```

It is generally preferable to change to a legal initial value rather than to weaken the declaration, but sometimes it is simpler to weaken the declaration than to try to make an initial value of the appropriate type.

Another declaration problem occasionally encountered is incorrect declarations on `defmacro` arguments. This probably usually happens when a function is converted into a macro. Consider this macro:

```
(defmacro my-1+ (x)
  (declare (fixnum x))
  `(the fixnum (1+ ,x)))
```

Although legal and well-defined COMMON LISP, this meaning of this definition is almost certainly not what the writer intended. For example, this call is illegal:

```
(my-1+ (+ 4 5))
```

The call is illegal because the argument to the macro is `(+ 4 5)`, which is a `list`, not a `fixnum`. Because of macro semantics, it is hardly ever useful to declare the types of macro arguments. If you really want to assert something about the type of the result of evaluating a macro argument, then put a `the` in the expansion:

```
(defmacro my-1+ (x)
  `(the fixnum (1+ (the fixnum ,x))))
```

In this case, it would be stylistically preferable to change this macro back to a function and declare it inline. Macros have no efficiency advantage over inline functions when using Python. See section 7.7.

Some more subtle problems are caused by incorrect declarations that can't be detected at compile time. Consider this code:

```
(do ((pos 0 (position #\a string :start (1+ pos))))
    ((null pos))
  (declare (fixnum pos))
  ...)
```

Although `pos` is almost always a `fixnum`, it is `nil` at the end of the loop. If this example is compiled with full type checks (the default), then running it will signal a type error at the end of the loop. If compiled without type checks, the program will go into an infinite loop (or perhaps `position` will complain because `(1+ nil)` isn't a sensible start.) Why? Because if you compile without type checks, the compiler just quietly believes the type declaration. Since `pos` is always a `fixnum`, it is never `nil`, so `(null pos)` is never true, and the loop exit test is optimized away. Such errors are sometimes flagged by unreachable code notes (section 7.4.5), but it is still important to initially compile any system with full type checks, even if the system works fine when compiled using other compilers.

In this case, the fix is to weaken the type declaration to `(or fixnum null)`.[6] Note that there is usually little performance penalty for weakening a declaration in this way. Any numeric operations in the body can still assume the variable is a `fixnum`, since `nil` is not a legal numeric argument. Another possible fix would be to say:

```
(do ((pos 0 (position #\a string :start (1+ pos))))
    ((null pos))
  (let ((pos pos))
    (declare (fixnum pos))
    ...))
```

This would be preferable in some circumstances, since it would allow a non-standard representation to be used for the local `pos` variable in the loop body (see section 7.9.3.)

In summary, remember that *all* values that a variable *ever* has must be of the declared type, and that you should test using safe code initially.

6.7. Compiler Policy

The policy is what tells the compiler *how* to compile a program. This is logically (and often textually) distinct from the program itself. Broad control of policy is provided by the `optimize` declaration; other declarations and variables control more specific aspects of compilation.

6.7.1. The Optimize Declaration

The `optimize` declaration recognizes six different *qualities*. The qualities are conceptually independent aspects of program performance. In reality, increasing one quality tends to have adverse effects on other qualities. The compiler compares the relative values of qualities when it needs to make a trade-off; i.e., if `speed` is greater than `safety`, then improve speed at the cost of safety.

The default for all qualities (except `debug-info`) is `1`. Whenever qualities are equal, ties are broken according to a broad idea of what a good default environment is supposed to be. Generally this downplays `speed`, `compile-speed` and `space` in favor of `safety` and `debug-info`. Novice and casual users should stick to

[6] Actually, this declaration is totally unnecessary in Python, since it already knows `position` returns a non-negative `fixnum` or `nil`.

the default policy. Advanced users often want to improve speed and memory usage at the cost of safety and debuggability.

If the value for a quality is 0 or 3, then it may have a special interpretation. A value of 0 means "totally unimportant", and a 3 means "ultimately important." These extreme optimization values enable "heroic" compilation strategies that are not always desirable and sometimes self-defeating. Specifying more than one quality as 3 is not desirable, since it doesn't tell the compiler which quality is most important.

These are the optimization qualities:

speed How fast the program should is run. **speed** 3 enables some optimizations that hurt debuggability.

compilation-speed
How fast the compiler should run. Note that increasing this above **safety** weakens type checking.

space How much space the compiled code should take up. Inline expansion is mostly inhibited when **space** is greater than **speed**. A value of 0 enables promiscuous inline expansion. Wide use of a 0 value is not recommended, as it may waste so much space that run time is slowed. See section 7.7 for a discussion of inline expansion.

debug-info How debuggable the program should be. The quality is treated differently from the other qualities: each value indicates a particular level of debugger information; it is not compared with the other qualities. See section 5.6 for more details.

safety How much error checking should be done. If **speed**, **space** or **compilation-speed** is more important than **safety**, then type checking is weakened (see section 6.5.3). If **safety** if 0, then no run time error checking is done. In addition to suppressing type checks, 0 also suppresses argument count checking, unbound-symbol checking and array bounds checks.

extensions:inhibit-warnings
This is a CMU extension that determines how little (or how much) diagnostic output should be printed during compilation. This quality is compared to other qualities to determine whether to print style notes and warnings concerning those qualities. If **speed** is greater than **inhibit-warnings**, then notes about how to improve speed will be printed, etc. The default value is 1, so raising the value for any standard quality above its default enables notes for that quality. If **inhibit-warnings** is 3, then all notes and most non-serious warnings are inhibited. This is useful with **declare** to suppress warnings about unavoidable problems.

6.8. Open Coding and Inline Expansion

Since COMMON LISP forbids the redefinition of standard functions[7], the compiler can have special knowledge of these standard functions embedded in it. This special knowledge is used in various ways (open coding, inline expansion, source transformation), but the implications to the user are basically the same:

- Attempts to redefine standard functions may be frustrated, since the function may never be called. Although it is technically illegal to redefine standard functions, users sometimes want to implicitly redefine these functions when they are debugging using the **trace** macro. Special-casing of standard functions can be inhibited using the **notinline** declaration.
- The compiler can have multiple alternate implementations of standard functions that implement different trade-offs of speed, space and safety. This selection is based on the compiler policy, see section 6.7.

When a function call is *open coded*, inline code whose effect is equivalent to the function call is substituted for

[7]See the proposed X3J13 "lisp-symbol-redefinition" cleanup.

that function call. When a function call is *closed coded*, it is usually left as is, although it might be turned into a call to a different function with different arguments. As an example, if **nthcdr** were to be open coded, then

```
(nthcdr 4 foobar)
```

might turn into

```
(cdr (cdr (cdr (cdr foobar))))
```

or even

```
(do ((i 0 (1+ i))
     (list foobar (cdr foobar)))
    ((= i 4) list))
```

If **nth** is closed coded, then

```
(nth x l)
```

might stay the same, or turn into something like:

```
(car (nthcdr x l))
```

In general, open coding sacrifices space for speed, but some functions (such as **car**) are so simple that they are always open-coded. Even when not open-coded, a call to a standard function may be transformed into a different function call (as in the last example) or compiled as *static call*. Static function call uses a more efficient calling convention that forbids redefinition.

Chapter 7

Advanced Compiler Use and Efficiency Hints

By Rob Maclachlan

7.1. Introduction

In CMU Common Lisp, as is any language on any computer, the path to efficient code starts with good algorithms and sensible programming techniques, but to avoid inefficiency pitfalls, you need to know some of this implementation's quirks and features. This chapter is mostly a fairly long and detailed overview of what optimizations Python does. Although there are the usual negative suggestions of inefficient features to avoid, the main emphasis is on describing the things that programmers can count on being efficient.

The optimizations described here can have the effect of speeding up existing programs written in conventional styles, but the potential for new programming styles that are clearer and less error-prone is at least as significant. For this reason, several sections end with a discussion of the implications of these optimizations for programming style.

7.1.1. Types

Python's support for types is unusual in three major ways:

- Precise type checking encourages the specific use of type declarations as a form of run-time consistency checking. This speeds development by localizing type errors and giving more meaningful error messages. See section 6.5.2. Python produces completely safe code; optimized type checking maintains reasonable efficiency on conventional hardware (see section 7.3.6.)
- Comprehensive support for the COMMON LISP type system makes complex type specifiers useful. Using type specifiers such as `or` and `member` has both efficiency and robustness advantages. See section 7.2.
- Type inference eliminates the need for some declarations, and also aids compile-time detection of type errors. Given detailed type declarations, type inference can often eliminate type checks and enable more efficient object representations and code sequences. Checking all types results in fewer type checks. See sections 7.3 and 7.9.2.

7.1.2. Optimization

The main barrier to efficient Lisp programs is not that there is no efficient way to code the program in Lisp, but that it is difficult to arrive at that efficient coding. Common Lisp is a highly complex language, and usually has many semantically equivalent "reasonable" ways to code a given problem. It is desirable to make all of these

equivalent solutions have comparable efficiency so that programmers don't have to waste time discovering the most efficient solution.

Source level optimization increases the number of efficient ways to solve a problem. This effect is much larger than the increase in the efficiency of the "best" solution. Source level optimization transforms the original program into a more efficient (but equivalent) program. Although the optimizer isn't doing anything the programmer couldn't have done, this high-level optimization is important because:

- The programmer can code simply and directly, rather than obfuscating code to please the compiler.
- When presented with a choice of similar coding alternatives, the programmer can chose whichever happens to be most convenient, instead of worrying about which is most efficient.

Source level optimization eliminates the need for macros to optimize their expansion, and also increases the effectiveness of inline expansion. See sections 7.4 and 7.7.

Efficient support for a safer programming style is the biggest advantage of source level optimization. Existing tuned programs typically won't benefit much from source optimization, since their source has already been optimized by hand. However, even tuned programs tend to run faster under Python because:

- Low level optimization and register allocation provides modest speedups in any program.
- Block compilation and inline expansion can reduce function call overhead, but may require some program restructuring. See sections 7.7 and 7.6
- Efficiency notes will point out important type declarations that are often missed even in highly tuned programs. See section 7.11.
- Existing programs can be compiled safely without prohibitive speed penalty, although they would be faster and safer with added declarations. See section 7.3.6.

7.1.3. Function Call

The sort of symbolic programs generally written in Common Lisp often favor recursion over iteration, or have inner loops so complex that they involve multiple function calls. Such programs spend a larger fraction of their time doing function calls than is the norm in other languages; for this reason Common Lisp implementations strive to make the general (or full) function call as inexpensive as possible. Python goes beyond this by providing two good alternatives to full call:

- Local call resolves function references at compile time, allowing better calling sequences and optimization across function calls. See section 7.6.
- Inline expansion totally eliminates call overhead and allows many context dependent optimizations. This provides a safe and efficient implementation of operations with function semantics, eliminating the need for error-prone macro definitions or manual case analysis. Although most COMMON LISP implementations support inline expansion, it becomes a more powerful tool with Python's source level optimization. See sections 7.4 and 7.7.

Generally, Python provides simple implementations for simple uses of function call, rather than having only a single calling convention. These features allow a more natural programming style:

- Proper tail recursion. See section 7.5
- Relatively efficient closures.
- A `funcall` that is as efficient as normal named call.
- Calls to local functions such as from `labels` are optimized:
  - Control transfer is a direct jump.
  - The closure environment is passed in registers rather than heap allocated.
  - Keyword arguments and multiple values are implemented more efficiently.

See section 7.6.

7.1.4. Object Representation

Sometimes traditional Common Lisp implementation techniques compare so poorly to the techniques used in other languages that Common Lisp can become an impractical language choice. Terrible inefficiencies appear in number-crunching programs, since Common Lisp numeric operations often involve number-consing and generic arithmetic. Python supports efficient natural representations for numbers (and some other types), and allows these efficient representations to be used in more contexts. Python also provides good efficiency notes that warn when a crucial declaration is missing.

See section 7.9.2 for more about object representations and numeric types. Also see section 7.11 about efficiency notes.

7.1.5. Writing Efficient Code

Writing efficient code that works is a complex and prolonged process. It is important not to get so involved in the pursuit of efficiency that you lose sight of what the original problem demands. Remember that:

- The program should be correct — it doesn't matter how quickly you get the wrong answer.
- Both the programmer and the user will make errors, so the program must be robust — it must detect errors in a way that allows easy correction.
- A small portion of the program will consume most of the resources, with the bulk of the code being virtually irrelevant to efficiency considerations. Even experienced programmers familiar with the problem area cannot reliably predict where these "hot spots" will be.

The best way to get efficient code that is still worth using, is to separate coding from tuning. During coding, you should:

- Use a coding style that aids correctness and robustness without being incompatible with efficiency.
- Choose appropriate data structures that allow efficient algorithms and object representations (see section 7.8). Try to make interfaces abstract enough so that you can change to a different representation if profiling reveals a need.
- Whenever you make an assumption about a function argument or global data structure, add consistency assertions, either with type declarations or explicit uses of **`assert`**, **`ecase`**, etc.

During tuning, you should:

- Identify the hot spots in the program through profiling (section 7.12.)
- Identify inefficient constructs in the hot spot with efficiency notes, more profiling, or manual inspection of the source. See sections 7.10 and 7.11.
- Add declarations and consider the application of optimizations. See sections 7.6, 7.7 and 7.9.2.
- If all else fails, consider algorithm or data structure changes. If you did a good job coding, changes will be easy to introduce.

7.2. More About Types in Python

This section goes into more detail describing what types and declarations are recognized by Python. The area where Python differs most radically from previous Common Lisp compilers is in its support for types:

- Precise type checking helps to find bugs at run time.
- Compile-time type checking helps to find bugs at compile time.
- Type inference minimizes the need for generic operations, and also increases the efficiency of run time

type checking and the effectiveness of compile time type checking.

- Support for detailed types provides a wealth of opportunity for operation-specific type inference and optimization.

7.2.1. More Types Meaningful

COMMON LISP has a very powerful type system, but conventional Common Lisp implementations typically only recognize the small set of types special in that implementation. In these systems, there is an unfortunate paradox: a declaration for a relatively general type like **fixnum** will be recognized by the compiler, but a highly specific declaration such as **(integer 3 17)** is totally ignored.

This is obviously a problem, since the user has to know how to specify the type of an object in the way the compiler wants it. A very minimal (but rarely satisfied) criterion for type system support is that it be no worse to make a specific declaration than to make a general one. Python goes beyond this by exploiting a number of advantages obtained from detailed type information.

Using more restrictive types in declarations allows the compiler to do better type inference and more compile-time type checking. Also, when type declarations are considered to be consistency assertions that should be verified (conditional on policy), then complex types are useful for making more detailed assertions.

Python "understands" the list-style **or**, **member**, **function**, array and number type specifiers. Understanding means that:

- If the type contains more information than is used in a particular context, then the extra information is simply ignored, rather than derailing type inference.
- In many contexts, the extra information from these type specifier is used to good effect. In particular, type checking in **Python** is *precise*, so these complex types can be used in declarations to make interesting assertions about functions and data structures (see section 6.5.2.) More specific declarations also aid type inference and reduce the cost for type checking.

For related information, see section 7.9 for numeric types, and section 7.8.3 for array types.

7.2.2. Canonicalization

When given a type specifier, Python will often rewrite it into a different (but equivalent) type. This is the mechanism that Python uses for detecting type equivalence. For example, in Python's canonical representation, these types are equivalent:

```
(or list (member :end)) <==> (or cons (member nil :end))
```

This has two implications for the user:

- The standard symbol type specifiers for **atom**, **null**, **fixnum**, etc., are in no way magical. The **null** type is actually defined to be **(member nil)**, **list** is **(or cons null)**, and **fixnum** is **(signed-byte 30)**.
- When the compiler prints out a type, it may not look like the type specifier that originally appeared in the program. This is generally not a problem, but it must be taken into consideration when reading compiler error messages.

7.2.3. Member Types

The **member** type specifier can be used to represent "symbolic" values, analogous to the enumerated types of Pascal. For example, the second value of **find-symbol** has this type:

```
(member :internal :external :inherited nil)
```

Member types are very useful for expressing consistency constraints on data structures, for example:

```
(defstruct ice-cream
  (flavor :vanilla :type (member :vanilla :chocolate :strawberry)))
```

Member types are also useful in type inference, as the number of members can sometimes be pared down to one, in which case the value is a known constant.

7.2.4. Union Types

The `or` (union) type specifier is understood, and is meaningfully applied in many contexts. The use of `or` allows assertions to be made about types in dynamically typed programs. For example:

```
(defstruct box
  (next nil :type (or box null))
  (top :removed :type (or box-top (member :removed))))
```

The type assertion on the `top` slot ensures that an error will be signalled when there is an attempt to store an illegal value (such as `:rmoved`.) Although somewhat weak, these union type assertions provide a useful input into type inference, allowing the cost of type checking to be reduced. For example, this loop is safely compiled with no type checks:

```
(defun find-box-with-top (box)
  (declare (type (or box null) box))
  (do ((current box (box-next current)))
      ((null current))
    (unless (eq (box-top current) :removed)
      (return current))))
```

Union types are also useful in type inference for representing types that are partially constrained. For example, the result of this expression:

```
(if foo
    (logior x y)
    (list x y))
```

can be expressed as `(or integer cons)`.

7.2.5. The Empty Type

The type `nil` is also called the empty type, since no object is of type `nil`. The union of no types, `(or)`, is also empty. Python's interpretation of an expression whose type is `nil` is that the expression never yields any value, but rather fails to terminate, or is thrown out of. For example, the type of a call to `error` or a use of `return` is `nil`. When the type of an expression is empty, compile-time type warnings about its value are suppressed; presumably somebody else is signalling an error. See also the function `required-argument` (page 73).

7.2.6. Function Types

`function` types are understood in the restrictive sense, specifying:

- The argument syntax that the function must be called with. This is information about what argument counts are acceptable, and which keyword arguments are recognized. In Python, warnings about argument syntax are a consequence of function type checking.
- The types of the argument values that the caller must pass. If the compiler can prove that some argument to a call is of a type disallowed by the called function's type, then it will give a compile-time type warning. In addition to being used for compile-time type checking, these type assertions are also used as output type assertions in code generation. For example, if `foo` is declared to have a `fixnum` argument, then the `1+` in `(foo (1+ x))` is compiled with knowledge that the result must be a fixnum.
- The types the values that will be bound to argument variables in the function's definition. Declaring a function's type with `ftype` implicitly declares the types of the arguments in the definition. Python checks for consistency between the definition and the `ftype` declaration. Because of precise type checking, an error will be signalled when a function is called with an argument of the wrong type.

- The type of return value(s) that the caller can expect. This information is a useful input to type inference. For example, if a function is declared to return a **fixnum**, then when a call to that function appears in an expression, the expression will be compiled with knowledge that the call will return a **fixnum**.
- The type of return value(s) that the definition must return. The result type in an **ftype** declaration is treated like an implicit **the** wrapped around the body of the definition. If the definition returns a value of the wrong type, an error will be signalled. If the compiler can prove that the function returns the wrong type, then it will give a compile-time warning.

This is consistent with the new interpretation of function types and the **ftype** declaration in the proposed X3J13 "function-type-argument-type-semantics" cleanup. Note also, that if you don't explicitly declare the type of a function using a global **ftype** declaration, then Python will compute a function type from the definition, providing a degree of inter-routine type inference, see section 7.3.3.

7.2.7. The Values Declaration

CMU Common Lisp supports the **values** declaration as an extension to COMMON LISP. The syntax is **(values** *type1 type2 ... typen*). This declaration is semantically equivalent to a **the** form wrapped around the body of the special form in which the **values** declaration appears. The advantage of **values** over **the** is that is purely syntactic — it doesn't introduce more indentation. For example:

```
(defun foo (x)
  (declare (values single-float))
  (ecase x
    (:this ...)
    (:that ...)
    (:the-other ...)))
```

is equivalent to:

```
(defun foo (x)
  (the single-float
       (ecase x
         (:this ...)
         (:that ...)
         (:the-other ...))))
```

and

```
(defun floor (number &optional (divisor 1))
  (declare (values integer real))
  ...)
```

is equivalent to:

```
(defun floor (number &optional (divisor 1))
  (the (values integer real)
       ...))
```

In addition to being recognized by **lambda** (and hence by **defun**), the **values** declaration is recognized by all the other special forms with bodies and declarations: **let**, **let***, **labels** and **flet**. Macros with declarations usually splice the declarations into one of the above forms, so they will accept this declaration too, but the exact effect of a **values** declaration will depend on the macro.

If you declare the types of all arguments to a function, and also declare the return value types with **values**, you have described the type of the function. Python will use this argument and result type information to derive a function type that will then be applied to calls of the function (see section 7.2.6.) This provides a way to declare the types of functions that is much less syntactically awkward than using the **ftype** declaration with a **function** type specifier.

Although the **values** declaration is non-standard, it is relatively harmless to use it in otherwise portable code, since any warning in non-CMU implementations can be suppressed with the standard **declaration**

proclamation.

7.2.8. Structure Types

Because of precise type checking, structure types are much better supported by Python than by conventional compilers:

- The structure argument to structure accessors is precisely checked — if you call `foo-a` on a `bar`, an error will be signalled.
- The types of slot values are precisely checked — if you pass the wrong type argument to a constructor or a slot setter, then an error will be signalled.

This error checking is tremendously useful for detecting bugs in programs that manipulate complex data structures.

An additional advantage of checking structure types and enforcing slot types is that the compiler can safely believe slot type declarations. Python effectively moves the type checking from the slot access to the slot setter or constructor call. This is more efficient since caller of the setter or constructor often knows the type of the value, entirely eliminating the need to check the value's type. Consider this example:

```
(defstruct coordinate
  (x nil :type single-float)
  (y nil :type single-float))

(defun make-it ()
  (make-coordinate 1.0 1.0))

(defun use-it (it)
  (declare (type coordinate it))
  (sqrt (expt (coordinate-x it) 2) (expt (coordinate-y it) 2)))
```

`make-it` and `use-it` are compiled with no checking on the types of the float slots, yet `use-it` can use `single-float` arithmetic with perfect safety. Note that `make-coordinate` must still check the values of `x` and `y` unless the call is block compiled or inline expanded (see section 7.6.) But even without this advantage, it is almost always more efficient to check slot values on structure initialization, since slots are usually written once and read many times.

7.2.9. The Freeze-Type Declaration

The `extensions:freeze-type` declaration is a CMU extension that enables more efficient compilation of user-defined types by asserting that the definition is not going to change. This declaration may only be used globally (with `declaim` or `proclaim`). Currently `freeze-type` only affects structure type testing done by `typep`, `typecase`, etc. Here is an example:

```
(declaim (freeze-type foo bar))
```

This asserts that the types `foo` and `bar` and their subtypes are not going to change. This allows more efficient type testing, since the compiler can open-code a test for all possible subtypes, rather than having to examine the type hierarchy at run-time.

7.2.10. Type Restrictions

Avoid use of the `and`, `not` and `satisfies` types in declarations, since type inference has problems with them. When these types do appear in a declaration, they are still checked precisely, but the type information is of limited use to the compiler. `and` types are effective as long as the intersection can be canonicalized to a type that doesn't use `and`. For example:

```
(and fixnum unsigned-byte)
```

is fine, since it is the same as:

```
(integer 0 most-positive-fixnum)
```

but this type:

```
(and symbol (not (member :end)))
```

will not be fully understood by type interference since the **and** can't be removed by canonicalization.

Using any of these type specifiers in a type test with **typep** or **typecase** is fine, since as tests, these types can be translated into the **and** macro, the **not** function or a call to the satisfies predicate.

7.2.11. Style Recommendations

Python provides good support for some currently unconventional ways of using the COMMON LISP type system. With Python, it is desirable to make declarations as precise as possible, but type inference also makes some declarations unnecessary. Here are some general guidelines for maximum robustness and efficiency:

- Declare the types of all function arguments and structure slots as precisely as possible (while avoiding **not**, **and** and **satisfies**). Put in these declarations during initial coding so that type assertions can find bugs for you during debugging.
- Use the **member** type specifier where there are a small number of possible symbol values, for example: `(member :red :blue :green)`.
- Use the **or** type specifier in situations where the type is not certain, but there are only a few possibilities, for example: `(or list vector)`.
- Declare integer types with the tightest bounds that you can, such as `(integer 3 7)`.
- Define **deftype** or **defstruct** types before they are used. Definition after use is legal (producing no "undefined type" warnings), but type tests and structure operations will be compiled much less efficiently.
- In addition to declaring the array element type and simpleness, also declare the dimensions if they are fixed, for example:

  ```
  (simple-array single-float (1024 1024))
  ```

  This bounds information allows array indexing for multi-dimensional arrays to be compiled much more efficiently, and may also allow array bounds checking to be done at compile time. See section 7.8.3.
- Avoid use of the **the** declaration within expressions. Not only does it clutter the code, but it is also almost worthless under safe policies. If the need for an output type assertion is revealed by efficiency notes during tuning, then you can consider **the**, but it is preferable to constrain the argument types more, allowing the compiler to prove the desired result type.
- Don't bother declaring the type of **let** or other non-argument variables unless the type is non-obvious. If you declare function return types and structure slot types, then the type of a variable is often obvious both to the programmer and to the compiler. An important case where the type isn't obvious, and a declaration is appropriate, is when the value for a variable is pulled out of untyped structure (e.g., the result of **car**), or comes from some weakly typed function, such as **read**.
- Declarations are sometimes necessary for integer loop variables, since the compiler can't always prove that the value is of a good integer type. These declarations are best added during tuning, when an efficiency note indicates the need.

7.3. Type Inference

Type inference is the process by which the compiler tries to figure out the types of expressions and variables, given an inevitable lack of complete type information. Although Python does much more type inference than most Common Lisp compilers, remember that the more precise and comprehensive type declarations are, the more type inference will be able to do.

7.3.1. Variable Type Inference

The type of a variable is the union of the types of all the definitions. In the degenerate case of a let, the type of the variable is the type of the initial value. This inferred type is intersected with any declared type, and is then propagated to all the variable's references. The types of `multiple-value-bind` variables are similarly inferred from the types of the individual values of the values form.

If multiple type declarations apply to a single variable, then all the declarations must be correct; it is as though all the types were intersected producing a single **and** type specifier. In this example:

```
(defmacro my-dotimes ((var count) &body body)
  `(do ((,var 0 (1+ ,var)))
       ((>= ,var ,count))
     (declare (type (integer 0 *) ,var))
     ,@body))

(my-dotimes (i ...)
  (declare (fixnum i))
  ...)
```

the two declarations for `i` are intersected, so `i` is known to be a non-negative fixnum.

In practice, this type inference is limited to lets and local functions, since the compiler can't analyze all the calls to a global function. But type inference works well enough on local variables so that it is often unnecessary to declare the type of local variables. This is especially likely when function result types and structure slot types are declared. The main areas where type inference breaks down are:

- When the initial value of a variable is a untyped expression, such as `(car x)`, and
- When the type of one of the variable's definitions is a function of the variable's current value, as in:

```
(setq x (1+ x))
```

7.3.2. Local Function Type Inference

The types of arguments to local functions are inferred in the same was as any other local variable; the type is the union of the argument types across all the calls to the function, intersected with the declared type. If there are any assignments to the argument variables, the type of the assigned value is unioned in as well.

The result type of a local function is computed in a special way that takes tail recursion (see section 7.5) into consideration. The result type is the union of all possible return values that aren't tail-recursive calls. For example, Python will infer that the result type of this function is `integer`:

```
(defun ! (n res)
  (declare (integer n res))
  (if (zerop n)
      res
      (! (1- n) (* n res))))
```

Although this is a rather obvious result, it becomes somewhat less trivial in the presence of mutual tail recursion of multiple functions. Local function result type inference interacts with the mechanisms for ensuring proper tail recursion mentioned in section 7.6.6.

7.3.3. Global Function Type Inference

As described in section 7.2.6, a global function type (`ftype`) declaration places implicit type assertions on the call arguments, and also guarantees the type of the return value. So wherever a call to a declared function appears, there is no doubt as to the types of the arguments and return value. Furthermore, Python will infer a function type from the function's definition if there is no `ftype` declaration. Any type declarations on the argument variables are used as the argument types in the derived function type, and the compiler's best guess for the result type of the

function is used as the result type in the derived function type.

This method of deriving function types from the definition implicitly assumes that functions won't be redefined at run-time. Consider this example:

```
(defun foo-p (x)
  (let ((res (and (consp x) (eq (car x) 'foo))))
    (format t "It is ~:[not ~;~]foo." res)))

(defun frob (it)
  (if (foo-p it)
      (setf (cadr it) 'yow!)
      (1+ it)))
```

Presumably, the programmer really meant to return `res` from `foo-p`, but he seems to have forgotten. When he tries to call do `(frob (list 'foo nil))`, `frob` will flame out when it tries to add to a `cons`. Realizing his error, he fixes `foo-p` and recompiles it. But when he retries his test case, he is baffled because the error is still there. What happened in this example is that Python proved that the result of `foo-p` is `null`, and then proceeded to optimize away the `setf` in `frob`.

Fortunately, in this example, the error is detected at compile time due to notes about unreachable code (see section 7.4.5.) Still, some users may not want to worry about this sort of problem during incremental development, so there is a variable to control deriving function types.

`extensions:*derive-function-types*` [*Variable*]
: If true (the default), argument and result type information derived from compilation of `defuns` is used when compiling calls to that function. If false, only information from `ftype` proclamations will be used.

7.3.4. Operation Specific Type Inference

Many of the standard COMMON LISP functions have special type inference procedures that determine the result type as a function of the argument types. For example, the result type of `aref` is the array element type. Here are some other examples of type inferences:

```
(logand x #xFF) ==> (unsigned-byte 8)

(+ (the (integer 0 12) x) (the (integer 0 1) y)) ==> (integer 0 13)

(ash (the (unsigned-byte 16) x) -8) ==> (unsigned-byte 8)
```

7.3.5. Dynamic Type Inference

Python uses flow analysis to infer types in dynamically typed programs. For example:

```
(ecase x
  (list (length x))
  ...)
```

Here, the compiler knows the argument to `length` is a list, because the call to `length` is only done when `x` is a list.

Dynamic type inference has two inputs: explicit conditionals and implicit or explicit type assertions. Flow analysis propagates these constraints on variable type to any code that can be executed only after passing though the constraint. Explicit type constraints come from `if`s where the test is either a lexical variable or a function of lexical variables and constants, where the function is either a type predicate, a numeric comparison or `eq`.

If there is an `eq` (or `eql`) test, then the compiler will actually substitute one argument for the other in the true

branch. For example:

```
(when (eq x :yow!) (return x))
```

becomes:

```
(when (eq x :yow!) (return :yow!))
```

This substitution is done when one argument is a constant, or one argument has better type information than the other. This transformation reveals opportunities for constant folding or type-specific optimizations. If the test is against a constant, then the compiler can prove that the variable is not that constant value in the false branch, or `(not (member :yow!))` in the example above. This can eliminate redundant tests, for example:

```
(if (eq x nil)
    ...
    (if x a b))
```

is transformed to this:

```
(if (eq x nil)
    ...
    a)
```

Variables appearing as `if` tests are interpreted as `(not (eq` *var* `nil))` tests. The compiler also converts = into `eql` where possible. It is difficult to do inference directly on = since it does implicit coercions.

When there is an explicit < or > test on integer variables, the compiler makes inferences about the ranges the variables can assume in the true and false branches. This is mainly useful when it proves that the values are small enough in magnitude to allow open-coding of arithmetic operations. For example, in many uses of `dotimes` with a `fixnum` repeat count, the compiler proves that fixnum arithmetic can be used.

Implicit type assertions are quite common, especially if you declare function argument types. Dynamic inference from implicit type assertions sometimes helps to disambiguate programs to a useful degree, but is most noticeable when it detects a dynamic type error. For example:

```
(defun foo (x)
  (+ (car x) x))
```

results in this warning:

```
In: DEFUN FOO
  (+ (CAR X) X)
==>
  X
Warning: Result is a LIST, not a NUMBER.
```

Note that Common Lisp's dynamic type checking semantics make dynamic type inference useful even in programs that aren't really dynamically typed, for example:

```
(+ (car x) (length x))
```

Here, `x` presumably always holds a list, but in the absence of a declaration the compiler cannot assume `x` is a list simply because list-specific operations are sometimes done on it. The compiler must consider the program to be dynamically typed until it proves otherwise. Dynamic type inference proves that the argument to `length` is always a list because the call to `length` is only done after the list-specific `car` operation.

The most significant efficiency effect of inference from assertions is usually in type check optimization.

7.3.6. Type Check Optimization

Python backs up its support for precise type checking by minimizing the cost of run-time type checking. This is done both through type inference and though optimizations of type checking itself.

Type inference often allows the compiler to prove that a value is of the correct type, and thus no type check is necessary. For example:

```
(defstruct foo a b c)
(defstruct link
  (foo (required-argument) :type foo)
  (next nil :type (or link null)))

(foo-a (link-foo x))
```

Here, there is no need to check that the result of `link-foo` is a `foo`, since it always is. Even when some type checks are necessary, type inference can often reduce the number:

```
(defun test (x)
  (let ((a (foo-a x))
        (b (foo-b x))
        (c (foo-c x)))
    ...))
```

In this example, only one `(foo-p x)` check is needed. This applies to a lesser degree in list operations, such as:

```
(if (eql (car x) 3) (cdr x) y)
```

Here, we only have to check that `x` is a list once.

Since Python recognizes explicit type tests, code that explicitly protects itself against type errors has little introduced overhead due to implicit type checking. For example, this loop compiles with no implicit checks checks for `car` and `cdr`:

```
(defun memq (e l)
  (do ((current l (cdr current)))
      ((atom current) nil)
    (when (eq (car current) e) (return current))))
```

Python reduces the cost of checks that must be done through an optimization called *complementing*. A complemented check for *type* is simply a check that the value is not of the type `(not` *type*`)`. This is only interesting when something is known about the actual type, in which case we can test for the complement of `(and` *known-type* `(not` *type*`))`, or the difference between the known type and the assertion. An example:

```
(link-foo (link-next x))
```

Here, we change the type check for `link-foo` from a test for `foo` to a test for:

```
(not (and (or foo null) (not foo)))
```

or more simply `(not null)`. This is probably the most important use of complementing, since the situation is fairly common, and a `null` test is much cheaper than a structure type test.

Here is a more complicated example that illustrates the combination of complementing with dynamic type inference:

```
(defun find-a (a x)
  (declare (type (or link null) x))
  (do ((current x (link-next current)))
      ((null current) nil)
    (let ((foo (link-foo current)))
      (when (eq (foo-a foo) a) (return foo)))))
```

This loop can be compiled with no type checks. The `link` test for `link-foo` and `link-next` is complemented to `(not null)`, and then deleted because of the explicit `null` test. As before, no check is necessary for `foo-a`, since the `link-foo` is always a `foo`. This sort of situation shows how precise type checking combined with precise declarations can actually result in reduced type checking.

7.4. Optimization

This section describes source-level transformations that Python does on programs in an attempt to make them more efficient. Although source-level optimizations can make existing programs more efficient, the biggest advantage of this sort of optimization is that it makes it easier to write efficient programs. If a clean, straightforward implementation is can be transformed into an efficient one, then there is no need for tricky and dangerous hand optimization.

7.4.1. Let Optimization

The primary optimization of let variables is to delete them when they are unnecessary. Whenever the value of a let variable is a constant, a constant variable or a constant (local) function, the variable is deleted, and references to the variable are replaced with references to the constant expression. This is useful primarily in the expansion of macros or inline functions, where argument values are often constant in any given call, but are in general non-constant expressions that must be bound to preserve order of evaluation. Let variable optimization eliminates the need for macros to carefully avoid spurious bindings, and also makes inline functions just as efficient as macros.

A particularly interesting class of constant is a local function. Substituting for lexical variables that are bound to a function can substantially improve the efficiency of functional programming styles, for example:

```
(let ((a #'(lambda (x) (zow x))))
  (funcall a 3))
```

effectively transforms to:

```
(zow 3)
```

This transformation is done even when the function is a closure, as in:

```
(let ((a (let ((y (zug)))
           #'(lambda (x) (zow x y)))))
  (funcall a 3))
```

becoming:

```
(zow 3 (zug))
```

A constant variable is a lexical variable that is never assigned to, always keeping its initial value. Whenever possible, avoid setting lexical variables — instead bind a new variable to the new value. Except for loop variables, it is almost always possible to avoid setting lexical variables. This form:

```
(let ((x (f x)))
  ...)
```

is *more* efficient than this form:

```
(setq x (f x))
...
```

Setting variables makes the program more difficult to understand, both to the compiler and to the programmer. Python compiles assignments at least as efficiently as any other Common Lisp compiler, but most let optimizations are only done on constant variables.

Constant variables with only a single use are also optimized away, even when the initial value is not constant.[8] For example, this expansion of `incf`:

```
(let ((#:g3 (+ x 1)))
  (setq x #:G3))
```

becomes:

```
(setq x (+ x 1))
```

[8]The source transformation in this example doesn't represent the preservation of evaluation order implicit in the compiler's internal representation. Where necessary, the back end will reintroduce temporaries to preserve the semantics.

The type semantics of this transformation are more important than the elimination of the variable itself. Consider what happens when `x` is declared to be a `fixnum`; after the transformation, the compiler can compile the addition knowing that the result is a `fixnum`, whereas before the transformation the addition would have to allow for fixnum overflow.

Another variable optimization deletes any variable that is never read. This causes the initial value and any assigned values to be unused, allowing those expressions to be deleted if they have no side-effects.

Note that a let is actually a degenerate case of local call (section 7.6.2), and that let optimization can be done on calls that weren't created by a let. Also, local call allows an applicative style of iteration that is totally assignment free.

7.4.2. Constant Folding

Constant folding is an optimization that replaces a call of constant arguments with the constant result of that call. Constant folding is done on all standard functions for which it is legal. Constant folding of a user defined function is enabled by the `extensions:constant-function` proclamation. Inline expansion allows folding of any constant parts of the definition, and can be done even on functions that have side-effects.

It is convenient to rely on constant folding when programming, as in this example:

```
(defconstant limit 42)

(defun foo ()
  (... (1- limit) ...))
```

Constant folding is also helpful when writing macros or inline functions, since it usually eliminates the need to write a macro that special-cases constant arguments.

7.4.3. Unused Expression Elimination

If the value of any expression is not used, and the expression has no side-effects, then it is deleted. As with constant folding, this optimization applies most often when cleaning up after inline expansion and other optimizations. Any function declared an `extensions:constant-function` is also subject to unused expression elimination.

Note that Python will eliminate parts of unused expressions known to be side-effect free, even if there are other unknown parts. For example:

```
(let ((a (list (foo) (bar))))
  (if t
      (zow)
      (raz a)))
```

becomes:

```
(progn (foo) (bar))
(zow)
```

7.4.4. Control Optimization

The most important optimization of control is recognizing when an `if` test is known at compile time, then deleting the `if`, the test expression, and the unreachable branch of the `if`. This can be considered a special case of constant folding, although the test doesn't have to be truly constant as long as it is definitely not `nil`. Note also, that type inference propagates the result of an `if` test to the true and false branches, see section 7.3.5.

A related `if` optimization is this transformation:[9]

```
(if (if a b c) x y)
```

into:

```
(if a
    (if b x y)
    (if c x y))
```

The opportunity for this sort of optimization usually results from a conditional macro. For example:

```
(if (not a) x y)
```

is actually implemented as this:

```
(if (if a nil t) x y)
```

which is transformed to this:

```
(if a
    (if nil x y)
    (if t x y))
```

which is then optimized to this:

```
(if a y x)
```

Note that due to Python's internal representations, the `if`—`if` situation will be recognized even if other forms are wrapped around the inner `if`, like:

```
(if (let ((g ...))
      (loop
        ...
        (return (not g))
        ...))
    x y)
```

In Python, all the COMMON LISP macros really are macros, written in terms of `if`, `block` and `tagbody`, so user-defined control macros can be just as efficient as the standard ones. Note that compiler emits basic blocks using a heuristic that minimizes the number of unconditional branches. The code in a `tagbody` will not be emitted in the order it appeared in the source, so there is no point in arranging the code to make control drop through to the target.

7.4.5. Unreachable Code Deletion

Python will delete code whenever it can prove that the code can never be executed. Code becomes unreachable when:

- An `if` is optimized away, or
- There is an explicit unconditional control transfer such as `go` or `return-from`, or
- The last reference to a local function is deleted (or there never was any reference.)

When code that appeared in the original source is deleted, the compiler prints a note to indicate a possible problem (or at least unnecessary code.) For example:

```
(defun foo ()
  (if t
      (write-line "True.")
      (write-line "False.")))
```

will result in this note:

[9]Note that the code for `x` and `y` isn't actually replicated.

```
In: DEFUN FOO
  (WRITE-LINE "False.")
Note: Deleting unreachable code.
```

It is important to pay attention to unreachable code notes, since they often indicate a subtle type error. For example:

```
(defstruct foo a b)

(defun lose (x)
  (let ((a (foo-a x))
        (b (if x (foo-b x) :none)))
    ...))
```

results in this note:

```
In: DEFUN LOSE
  (IF X (FOO-B X) :NONE)
==>
  :NONE
Note: Deleting unreachable code.
```

The `:none` is unreachable, because type inference knows that the argument to `foo-a` must be a `foo`, and thus can't be `nil`. Presumably the programmer forgot that `x` could be `nil` when he wrote the binding for `a`.

Here is an example with an incorrect declaration:

```
(defun count-a (string)
  (do ((pos 0 (position #\a string :start (1+ pos)))
       (count 0 (1+ count)))
      ((null pos) count)
    (declare (fixnum pos))))
```

This time our note is:

```
In: DEFUN COUNT-A
  (DO ((POS 0 #) (COUNT 0 #))
      ((NULL POS) COUNT)
    (DECLARE (FIXNUM POS)))
--> BLOCK LET TAGBODY RETURN-FROM PROGN
==>
  COUNT
Note: Deleting unreachable code.
```

The problem here is that `pos` can never be null since it is declared a `fixnum`.

It takes some experience with unreachable code notes to be able to tell what they are trying to say. In non-obvious cases, the best thing to do is to call the function in a way that should cause the unreachable code to be executed. Either you will get a type error, or you will find that there truly is no way for the code to be executed.

Not all unreachable code results in a note:

- A note is only given when the unreachable code textually appears in the original source. This prevents spurious notes due to the optimization of macros and inline functions, but sometimes also foregoes a note that would have been useful.
- Since accurate source information is not available for non-list forms, there is an element of heuristic in determining whether or not to give a note about an atom. Spurious notes may be given when a macro or inline function defines a variable that is also present in the calling function. Notes about `nil` and `t` are never given, since it is too easy to confuse these constants in expanded code with ones in the original source.
- Notes are only given about code unreachable due to control flow. There is no note when an expression is deleted because its value is unused, since this is a common consequence of other optimizations.

Somewhat spurious unreachable code notes can also result when a macro inserts multiple copies of its arguments in different contexts, for example:

```
(defmacro t-and-f (var form)
  `(if ,var ,form ,form))

(defun foo (x)
  (t-and-f x (if x "True." "False.")))
```

results in these notes:

```
In: DEFUN FOO
  (IF X "True." "False.")
==>
  "False."
Note: Deleting unreachable code.

==>
  "True."
Note: Deleting unreachable code.
```

It seems like it has deleted both branches of the `if`, but it has really deleted one branch in one copy, and the other branch in the other copy. Note that these messages are only spurious in not satisfying the intent of the rule that notes are only given when the deleted code appears in the original source; there is always *some* code being deleted when a unreachable code note is printed.

7.4.6. Multiple Values Optimization

Within a function, Python implements uses of multiple values particularly efficiently. Multiple values can be kept in arbitrary registers, so using multiple values doesn't imply stack manipulation and representation conversion. For example, this code:

```
(let ((a (if x (foo x) u))
      (b (if x (bar x) v)))
  ...)
```

is actually more efficient written this way:

```
(multiple-value-bind
    (a b)
    (if x
        (values (foo x) (bar x))
        (values u v))
  ...)
```

Also, see section 7.6.6 for information on how local call provides efficient support for multiple function return values.

7.4.7. Source to Source Transformation

The compiler implements a number of operation-specific optimizations as source-to-source transformations. You will often see unfamiliar code in error messages, for example:

```
(defun my-zerop () (zerop x))
```

gives this warning:

```
In: DEFUN MY-ZEROP
  (ZEROP X)
==>
  (= X 0)
Warning: Undefined variable: X
```

The original `zerop` has been transformed into a call to `=`. This transformation is indicated with the same ==> used

to mark macro and function inline expansion. Although it can be confusing, display of the transformed source is important, since warnings are given with respect to the transformed source. This a more obscure example:

```
(defun foo (x) (logand 1 x))
```

gives this efficiency note:

```
In: DEFUN FOO
  (LOGAND 1 X)
==>
  (LOGAND C::Y C::X)
Note: Forced to do static-function Two-arg-and (cost 53).
      Unable to do inline fixnum arithmetic (cost 1) because:
      The first argument is a INTEGER, not a FIXNUM.
      etc.
```

Here, the compiler commuted the call to **logand**, introducing temporaries. The note complains that the *first* argument is not a **fixnum**, when in the original call, it was the second argument. To make things more confusing, the compiler introduced temporaries called **c::x** and **c::y** that are bound to **y** and **1**, respectively.

You will also notice source-to-source optimizations when efficiency notes are enabled (see section 7.11.) When the compiler is unable to do a transformation that might be possible if there was more information, then an efficiency note is printed. For example, **my-zerop** above will also give this efficiency note:

```
In: DEFUN FOO
  (ZEROP X)
==>
  (= X 0)
Note: Unable to optimize because:
      Operands might not be the same type, so can't open code.
```

7.4.8. Style Recommendations

Source level optimization makes possible a clearer and more relaxed programming style:

- Don't use macros purely to avoid function call. If you want an inline function, write it as a function and declare it inline. It's clearer, less error-prone, and works just as well.
- Don't write macros that try to "optimize" their expansion in trivial ways such as avoiding binding variables for simple expressions. The compiler does these optimizations too, and is less likely to make a mistake.
- Make use of local functions (i.e., **labels** or **flet**) and tail-recursion in places where it is clearer. Local function call is faster than full call.
- Avoid setting local variables when possible. Binding a new **let** variable is at least as efficient as setting an existing variable, and is easier to understand, both for the compiler and the programmer.
- Instead of writing similar code over and over again so that it can be hand customized for each use, define a macro or inline function, and let the compiler do the work.

7.5. Tail Recursion

A call is tail-recursive if nothing has to be done after the the call returns, i.e. when the call returns, the returned value is immediately returned from the calling function. In this example, the recursive call to **myfun** is tail-recursive:

```
(defun myfun (x)
  (if (oddp (random x))
      (isqrt x)
      (myfun (1- x))))
```

Tail recursion is interesting because it is form of recursion that can be implemented much more efficiently than general recursion. In general, a recursive call requires the compiler to allocate storage on the stack at run-time for every call that has not yet returned. This memory consumption makes recursion unacceptably inefficient for representing repetitive algorithms having large or unbounded size. Tail recursion is the special case of recursion that is semantically equivalent to the iteration constructs normally used to represent repetition in programs. Because tail recursion is equivalent to iteration, tail-recursive programs can be compiled as efficiently as iterative programs.

So why would you want to write a program recursively when you can write it using a loop? Well, the main answer is that recursion is a more general mechanism, so it can express some solutions simply that are awkward to write as a loop. Some programmers also feel that recursion is a stylistically preferable way to write loops because it avoids assigning variables. For example, instead of writing:

```
(defun fun1 (x)
  something-that-uses-x)

(defun fun2 (y)
  something-that-uses-y)

(do ((x something (fun2 (fun1 x))))
    (nil))
```

You can write:

```
(defun fun1 (x)
  (fun2 something-that-uses-x))

(defun fun2 (y)
  (fun1 something-that-uses-y))

(fun1 something)
```

The tail-recursive definition is actually more efficient, in addition to being (arguably) clearer. As the number of functions and the complexity of their call graph increases, the simplicity of using recursion becomes compelling. Consider the advantages of writing a large finite-state machine with separate tail-recursive functions instead of using a single huge `prog`.

It helps to understand how to use tail recursion if you think of a tail-recursive call as a `psetq` that assigns the argument values to the called function's variables, followed by a `go` to the start of the called function. This makes clear an inherent efficiency advantage of tail-recursive call: in addition to not having to allocate a stack frame, there is no need to prepare for the call to return (e.g., by computing a return PC.)

Is there any disadvantage to tail recursion? Other than an increase in efficiency, the only way you can tell that a call has been compiled tail-recursively is if you use the debugger. Since a tail-recursive call has no stack frame, there is no way the debugger can print out the stack frame representing the call. The effect is that backtrace will not show some calls that would have been displayed in a non-tail-recursive implementation. In practice, this is not as bad as it sounds — in fact it isn't really clearly worse, just different. See section 5.3.5 for information about the debugger implications of tail recursion.

In order to ensure that tail-recursion is preserved in arbitrarily complex calling patterns across separately compiled functions, the compiler must compile any call in a tail-recursive position as a tail-recursive call. This is done regardless of whether the program actually exhibits any sort of recursive calling pattern. In this example, the call to `fun2` will always be compiled as a tail-recursive call:

```
(defun fun1 (x)
  (fun2 x))
```

So tail recursion doesn't necessarily have anything to do with recursion as it is normally thought of.

See section 7.6.4 for more discussion of using tail recursion to implement loops.

7.5.1. Tail Recursion Exceptions

Although Python is claimed to be "properly" tail-recursive, some might dispute this, since there are situations where tail recursion is inhibited:

- When the call is enclosed by a special binding, or
- When the call is enclosed by a **catch** or **unwind-protect**, or
- When the call is enclosed by a **block** or **tagbody** and the block name or **go** tag has been closed over.

These dynamic extent binding forms inhibit tail recursion because they allocate stack space to represent the binding. Shallow-binding implementations of dynamic scoping also require cleanup code to be evaluated when the scope is exited.

7.6. Local Call and Block Compilation

Python supports two kinds of function call: full call and local call. Full call is the standard calling convention; its late binding and generality make Common Lisp what it is, but create unavoidable overheads. When the compiler can compile the calling function and the called function simultaneously, it can use local call to avoid some of the overhead of full call. Local call is really a collection of compilation strategies. If some aspect of call overhead is not needed in a particular local call, then it can be omitted. In some cases, local call can be totally free. Local call provides two main advantages to the user:

- Local call makes the use of the lexical function binding forms **flet** and **labels** much more efficient. A local call is always faster than a full call, and in many cases is much faster.
- Local call is a natural approach to *block compilation*, a compilation technique that resolves function references at compile time. Block compilation speeds function call, but increases compilation times and prevents function redefinition.

7.6.1. Self-Recursive Calls

Local call is used when a function defined by **defun** calls itself. For example:

```
(defun fact (n)
  (if (zerop n)
      1
      (* n (fact (1- n)))))
```

This use of local call speeds recursion, but can also complicate debugging, since **trace** will only show the first call to **fact**, and not the recursive calls. This is because the recursive calls directly jump to the start of the function, and don't indirect through the **symbol-function**.

7.6.2. Let Calls

Because local call avoids unnecessary call overheads, the compiler internally uses local call to implement some macros and special forms that are not normally thought of as involving a function call. For example, this **let**:

```
(let ((a (foo))
      (b (bar)))
  ...)
```

is internally represented as though it was macroexpanded into:

```
(funcall #'(lambda (a b)
             ...)
         (foo)
         (bar))
```

This implementation is acceptable because the simple cases of local call (equivalent to a `let`) result in good code. This doesn't make `let` any more efficient, but does make local calls that are semantically the same as `let` much more efficient than full calls. For example, these definitions are all the same as far as the compiler is concerned:

```
(defun foo ()
  ...some other stuff...
  (let ((a something))
    ...some stuff...))

(defun foo ()
  (flet ((localfun (a)
           ...some stuff...))
    ...some other stuff...
    (localfun something)))

(defun foo ()
  (let ((funvar #'(lambda (a)
                    ...some stuff...)))
    ...some other stuff...
    (funcall funvar something)))
```

Although local call is most efficient when the function is called only once, a call doesn't have to be equivalent to a `let` to be more efficient than full call. All local calls avoid the overhead of argument count checking and keyword argument parsing, and there are a number of other advantages that apply in many common situations. See also section 7.4.1 for a discussion of the optimizations done on let calls.

7.6.3. Closures

Local call allows for much more efficient use of closures, since the closure environment doesn't need to be allocated on the heap, or even stored in memory at all. In this example, there is no penalty for `localfun` referencing `a` and `b`:

```
(defun foo (a b)
  (flet ((localfun (x)
           (1+ (* a b x))))
    (if (= a b)
        (localfun (- x))
        (localfun x))))
```

In local call, the compiler effectively passes closed-over values as extra arguments, so there is no need for you to "optimize" local function use by explicitly passing in lexically visible values. Closures may also be subject to let optimization (see section 7.4.1.)

Note: currently indirect value cells are always allocated on the heap when a variable is both assigned to (with `setq` or `setf`) and closed over, regardless of whether the closure is a local function or not. This is another reason to avoid setting variables when you don't have to.

7.6.4. Tail Recursion

Tail-recursive local calls are particularly efficient, since they are in effect an assignment plus a control transfer. Scheme programmers write loops with tail-recursive local calls, instead of using the imperative `go` and `setq`. This has not caught on in the COMMON LISP community, since conventional Common Lisp compilers don't implement local call. In Python, users can choose to write loops such as:

```
(defun ! (n)
  (labels ((loop (n total)
             (if (zerop n)
                 total
                 (loop (1- n) (* n total)))))
    (loop n 1)))
```

extensions:iterate *name* ({(*var initial-value*)}*) {*declaration*}* {*form*}* [*Macro*]

This macro provides syntactic sugar for using **labels** to do iteration. It creates a local function *name* with the specified *vars* as its arguments and the *declarations* and *forms* as its body. This function is then called with the *initial-values*, and the result of the call is return from the macro.

Here is our factorial example rewritten using **iterate**:

```
(defun ! (n)
  (iterate loop
           ((n n)
            (total 1))
    (if (zerop n)
        total
        (loop (1- n) (* n total)))))
```

The main advantage of using **iterate** over **do** is that **iterate** naturally allows stepping to be done differently depending on conditionals in the body of the loop. **iterate** can also be used to implement algorithms that aren't really iterative by simply doing a non-tail call. For example, the standard recursive definition of factorial can be written like this:

```
(iterate fact
         ((n n))
  (if (zerop n)
      1
      (* n (fact (1- n)))))
```

7.6.5. Block Compilation

Block compilation allows calls to global functions defined by **defun** to be compiled as local calls. The function call can be in a different top-level form than the **defun**, or even in a different file. Block compilation is enabled by specifying **t** for the **:block-compile** argument to **compile-file**.

The effect of block compilation can be envisioned by imagining that the compiler turns all the **defuns** in the file into a single **labels** form:

```
(defun fun1 ()
  ...)

(defun fun2 ()
  ...
  (fun1)
  ...)

(defun fun3 (x)
  (if x
      (fun1)
      (fun2)))
```

becomes:

```
(labels ((fun1 ()
           ...)
         (fun2 ()
           ...
           (fun1)
           ...)
         (fun3 (x)
           (if x
               (fun1)
               (fun2))))
  (setf (symbol-function 'fun1) #'fun1)
  (setf (symbol-function 'fun2) #'fun2)
  (setf (symbol-function 'fun3) #'fun2))
```

Block compilation still installs global definitions of the functions in the `symbol-function`, and a full call to a block-compiled function works just as before. Calls between the block compiled functions are local calls, so changing the global definition of `fun1` will have no effect on what `fun2` does; `fun2` will keep calling the old `fun1`.

The main problem with block compilation is that the compiler uses large amounts of memory when it is block compiling. This places an upper limit on the size of file that can be block compiled. It is helpful to break off the important parts of a program into a file small enough to be block compiled. Unless files are very small, it is probably impractical to block compile files together by specifying a list of files to `compile-file`. Semi-inline expansion (section 7.7.2) provides another way to extend block compilation across file boundaries.

7.6.6. Return Values

One of the more subtle costs of full call comes from allowing arbitrary numbers of return values. This overhead can be avoided in local calls to functions that always return the same number of values. For efficiency reasons (as well as stylistic ones), you should write functions so that they always return the same number of values. This may require passing extra `nil` arguments to `values` in some cases, but the result is more efficient, not less so.

When efficiency notes are enabled (section 7.11), and the compiler wants to use known values return, but can't prove that the function always returns the same number of values, then it will print a note like this:

```
In: DEFUN GRUE
  (DEFUN GRUE (X) (DECLARE (FIXNUM X)) (COND (# #) (# NIL) (T #)))
Note: Return type not fixed values, so can't use known return convention:
  (VALUES (OR (INTEGER -536870912 -1) NULL) &REST T)
```

In order to implement proper tail recursion in the presence of known values return (section 7.5), the compiler sometimes must prove that multiple functions all return the same number of values. When this can't be proven, the compiler will print a note like this:

```
In: DEFUN BLUE
  (DEFUN BLUE (X) (DECLARE (FIXNUM X)) (COND (# #) (# #) (# #) (T #)))
Note: Return value count mismatch prevents known return from
      these functions:
  BLUE
  SNOO
```

You probably won't see this note very often.

7.7. Inline Expansion

Python can expand almost any function inline, including functions with keyword arguments. The only restrictions are that there can't be a `&rest` argument, and the keywords in the call to a function with keyword arguments must be constant. Combined with Python's source-level optimization, inline expansion can be used for things that formerly required macros for efficient implementation. In Python, macros don't have any efficiency advantage, so they need only be used where a macro's syntactic flexibility is required.

Inline expansion is a compiler optimization technique that reduces the overhead of a function call by simply not doing the call: instead, the compiler effectively rewrites the program to appear as though the definition of the called function was inserted at each call site. In Common Lisp, this is straightforwardly expressed by inserting the `lambda` corresponding to the original definition:

```
(proclaim '(inline my-1+))
(defun my-1+ (x) (+ x 1))

(my-1+ someval) ==> ((lambda (x) (+ x 1)) someval)
```

When the function expanded inline is large, the program after inline expansion may be substantially larger than the original program. If the program becomes too large, inline expansion hurts speed rather than helping it, since hardware resources such as physical memory and cache will be exhausted. Inline expansion is called for:

- When profiling has shown that a relatively simple function is called so often that a large amount of time is being wasted in the calling of that function (as opposed to running in that function.) If a function is complex, it will take a long time to run relative the time spent in call, so the speed advantage of inline expansion is diminished at the same time the space cost of inline expansion is increased. Of course, if a function is rarely called, then the overhead of calling it is also insignificant.
- With functions so simple that they take less space to inline expand than would be taken to call the function (such as `my-1+` above.) It would require intimate knowledge of the compiler to be certain when inline expansion would reduce space, but it is generally safe to inline expand functions whose definition is a single function call, or a few calls to simple COMMON LISP functions.

In addition to this speed/space tradeoff from inline expansion's avoidance of the call, inline expansion can also reveal opportunities for optimization. Python's extensive source-level optimization can make use of context information from the caller to tremendously simplify the code resulting from the inline expansion of a function.

The main form of caller context is local information about the actual argument values: what the argument types are and whether the arguments are constant. Knowledge about argument types can eliminate run-time type tests (e.g., for generic arithmetic.) Constant arguments in a call provide opportunities for constant folding optimization after inline expansion.

A hidden way that constant arguments are often supplied to functions is through the defaulting of unsupplied optional or keyword arguments. There can be a huge efficiency advantage to inline expanding functions that have complex keyword-based interfaces, such as this definition of the `member` function:

```
(proclaim '(inline member))
(defun member (item list &key
                    (key #'identity)
                    (test #'eql testp)
                    (test-not nil notp))
  (do ((list list (cdr list)))
      ((null list) nil)
    (let ((car (car list)))
      (if (cond (testp
                 (funcall test item (funcall key car)))
                (notp
                 (not (funcall test-not item (funcall key car))))
                (t
                 (funcall test item (funcall key car))))
          (return list)))))
```

After inline expansion, this call is simplified to the obvious code:

```
(member a l :key #'foo-a :test #'char=) ==>

(do ((list list (cdr list)))
    ((null list) nil)
  (let ((car (car list)))
    (if (char= item (foo-a car))
        (return list))))
```

In this example, there could easily be more than an order of magnitude improvement in speed. In addition to eliminating the original call to **member**, inline expansion also allows the calls to **char=** and **foo-a** to be open-coded. We go from a loop with three tests and two calls to a loop with one test and no calls.

See section 7.4 for more discussion of source level optimization.

7.7.1. Inline Expansion Recording

Inline expansion requires that the source for the inline expanded function to be available when calls to the function are compiled. The compiler doesn't remember the inline expansion for every function, since that would take an excessive about of space. Instead, the programmer must tell the compiler to record the inline expansion before the definition of the inline expanded function is compiled. This is done by globally declaring the function inline before the function is defined, by using the **inline** and **extensions:maybe-inline** (see section 7.7.3) declarations.

In addition to recording the inline expansion of inline functions at the time the function is compiled, **compile-file** also puts the inline expansion in the output file. When the output file is loaded, the inline expansion is made available for subsequent compilations; there is no need to compile the definition again to record the inline expansion.

If a function is declared inline, but no expansion is recorded, then the compiler will give an efficiency note like:

```
Note: MYFUN is declared inline, but has no expansion.
```

When you get this note, check that the **inline** declaration and the definition appear before the calls that are to be inline expanded.

7.7.2. Semi-Inline Expansion

Python supports *semi-inline* functions. Semi-inline expansion shares a single copy of a function across all the calls in a component by converting the inline expansion into a local function (see section 7.6.) This takes up less space when there are multiple calls, but also provides less opportunity for context dependent optimization. When

there is only one call, the result is identical to normal inline expansion. Semi-inline expansion is done when the **space** optimization quality is 0, and the function has been declared **extensions:maybe-inline**.

This mechanism of inline expansion combined with local call also allows recursive functions to be inline expanded. If a recursive function is declared **inline**, calls will actually be compiled semi-inline. Although recursive functions are often so complex that there is little advantage to semi-inline expansion, it can still be useful in the same sort of cases where normal inline expansion is especially advantageous, i.e. functions where the calling context can help a lot.

7.7.3. The Maybe-Inline Declaration

The **extensions:maybe-inline** declaration is a CMU Common Lisp extension. It is similar to **inline**, but indicates that inline expansion may sometimes be desirable, rather than saying that inline expansion should almost always be done. When used in a global declaration, **extensions:maybe-inline** causes the expansion for the named functions to be recorded, but the functions aren't actually inline expanded unless **space** is 0 or the function is eventually (perhaps locally) declared **inline**.

Use of the **extensions:maybe-inline** declaration followed by the **defun** is preferable to the standard idiom of:

```
(proclaim '(inline myfun))
(defun myfun () ...)
(proclaim '(notinline myfun))

;;; Any calls to myfun here are not inline expanded.

(defun somefun ()
  (declare (inline myfun))
  ;;
  ;; Calls to myfun here are inline expanded.
  ...)
```

The problem with using **notinline** in this way is that in COMMON LISP it does more than just suppress inline expansion, it also forbids the compiler to use any knowledge of **myfun** until a later **inline** declaration overrides the **notinline**. This prevents compiler warnings about incorrect calls to the function, and also prevents block compilation.

The **extensions:maybe-inline** declaration is used like this:

```
(proclaim '(extensions:maybe-inline myfun))
(defun myfun () ...)

;;; Any calls to myfun here are not inline expanded.

(defun somefun ()
  (declare (inline myfun))
  ;;
  ;; Calls to myfun here are inline expanded.
  ...)

(defun someotherfun ()
  (declare (optimize (space 0)))
  ;;
  ;; Calls to myfun here are expanded semi-inline.
  ...)
```

In this example, the use of **extensions:maybe-inline** causes the expansion to be recorded when the **defun** for **somefun** is compiled, and doesn't waste space through doing inline expansion by default. Unlike

notinline, this declaration still allows the compiler to assume that the known definition really is the one that will be called when giving compiler warnings, and also allows the compiler to do semi-inline expansion when the policy is appropriate.

When the goal is merely to control whether inline expansion is done by default, it is preferable to use **extensions:maybe-inline** rather than **notinline**. The **notinline** declaration should be reserved for those special occasions when a function may be redefined at run-time, so the compiler must be told that the obvious definition of a function is not necessarily the one that will be in effect at the time of the call.

7.8. Object Representation

A somewhat subtle aspect of writing efficient COMMON LISP programs is choosing the correct data structures so that the underlying objects can be implemented efficiently. This is partly because of the need for multiple representations for a given value (see section 7.9.2), but is also due to the sheer number of object types that COMMON LISP has built in. The number of possible representations complicates the choice of a good representation because semantically similar objects may vary in their efficiency depending on how the program operates on them.

7.8.1. Think Before You Use a List

Although Lisp's creator seemed to think that it was for LISt Processing, the astute observer may have noticed that the chapter on list manipulation makes up less that three percent of *Common Lisp: the Language II*. The language has grown since Lisp 1.5 — new data types supersede lists for many purposes.

7.8.2. Structures

One of the best ways of building complex data structures is to define appropriate structure types using **defstruct**. In Python, access of structure slots is always at least as fast as list or vector access, and is usually faster. In comparison to a list representation of a tuple, structures also have a space advantage.

Even if structures weren't more efficient than other representations, structure use would still be attractive because programs that use structures in appropriate ways are much more maintainable and robust than programs written using only lists. For example:

```
(rplaca (caddr (cadddr x)) (caddr y))
```

could have been written using structures in this way:

```
(setf (beverage-flavor (astronaut-beverage x)) (beverage-flavor y))
```

The second version is more maintainable because it is easier to understand what it is doing. It is more robust because structures accesses are type checked. An **astronaut** will never be confused with a **beverage**, and the result of **beverage-flavor** is always a flavor. See sections 7.2.8 and 7.2.9 for more information about structure types. See section 7.3 for a number of examples that make clear the advantages of structure typing.

Note that the structure definition should be compiled before any uses of its accessors or type predicate so that these function calls can be efficiently open-coded.

7.8.3. Arrays

Arrays are often the most efficient representation for collections of objects because:

- Array representations are often the most compact. An array is always more compact than a list containing the same number of elements.
- Arrays allow fast constant-time access.
- Arrays are easily destructively modified, which can reduce consing.

- Array element types can be specialized, which reduces both overall size and consing (see section 7.9.8.)

Access of arrays that are not of type **simple-array** is less efficient, so declarations are appropriate when an array is of a simple type like **simple-string** or **simple-bit-vector**. Arrays are almost always simple, but the compiler may not be able to prove simpleness at every use. The only way to get a non-simple array is to use the **:displaced-to**, **:fill-pointer** or **adjustable** arguments to **make-array**. If you don't use these hairy options, then arrays can always be declared to be simple.

Because of the many specialized array types and the possibility of non-simple arrays, array access is much like generic arithmetic (section 7.9.4). In order for array accesses to be efficiently compiled, the element type and simpleness of the array must be known at compile time. If there is inadequate information, the compiler is forced to call a generic array access routine. You can detect inefficient array accesses by enabling efficiency notes, see section 7.11.

7.8.4. Vectors

Vectors (one dimensional arrays) are particularly useful, since in addition to their obvious array-like applications, they are also well suited to representing sequences. In comparison to a list representation, vectors are faster to access and take up between two and sixty-four times less space (depending on the element type.) As with arbitrary arrays, the compiler needs to know that vectors are not complex, so you should use **simple-string** in preference to **string**, etc.

The only advantage that lists have over vectors for representing sequences is that it is easy to change the length of a list, add to it and remove items from it. Likely signs of archaic, slow lisp code are **nth** and **nthcdr**. If you are using these functions you should probably be using a vector.

7.8.5. Bit-Vectors

Another thing that lists have been used for is set manipulation. In applications where there is a known, reasonably small universe of items bit-vectors can be used to improve performance. This is much less convenient than using lists, because instead of symbols, each element in the universe must be assigned a numeric index into the bit vector. Using a bit-vector will nearly always be faster, and can be tremendously faster if the number of elements in the set is not small. The logical operations on **simple-bit-vectors** are efficient, since they operate on a word at a time.

7.8.6. Hashtables

Hashtables are an efficient and general mechanism for maintaining associations such as the association between an object and its name. Although hashtables are usually the best way to maintain associations, efficiency and style considerations sometimes favor the use of an association list (a-list).

assoc is fairly fast when the *test* argument is **eq** or **eql** and there are only a few elements, but the time goes up in proportion with the number of elements. In contrast, the hash-table lookup has a somewhat higher overhead, but the speed is largely unaffected by the number of entries in the table. For an **equal** hash-table or alist, hash-tables have an even greater advantage, since the test is more expensive. Whatever you do, be sure to use the most restrictive test function possible.

The style argument observes that although hash-tables and alists overlap in function, they do not do all things equally well.

- Alists are good for maintaining scoped environments. They were originally invented to implement scoping in the Lisp interpreter, and are still used for this in Python. With an alist one can non-destructively change an association simply by consing a new element on the front. This is something that cannot be done with hash-tables.

- Hashtables are good for maintaining a global association. The value associated with an entry can easily be changed with `setf`. With an alist, one has to go through contortions, either `rplacd`'ing the cons if the entry exists, or pushing a new one if it doesn't. The side-effecting nature of hash-table operations is an advantage here.

Historically, symbol property lists were often used for global name associations. Property lists provide an awkward and error-prone combination of name association and record structure. If you must use the property list, please store all the related values in a single structure under a single property, rather than using many properties. This makes access more efficient, and also adds a modicum of typing and abstraction. See section 7.2 for information on types in CMU Common Lisp.

7.9. Numbers

Numbers are interesting because numbers are one of the few Common Lisp data types that have direct support in conventional hardware. If a number can be represented in the way that the hardware expects it, then there is a big efficiency advantage.

Using hardware representations is problematical in Common Lisp due to dynamic typing (where the type of a value may be unknown at compile time.) It is possible to compile code for statically typed portions of a Common Lisp program with efficiency comparable to that obtained in statically typed languages such as C, but not all Common Lisp implementations succeed. There are two main barriers to efficient numerical code in Common Lisp:

- The compiler must prove that the numerical expression is in fact statically typed, and
- The compiler must be able to somehow reconcile the conflicting demands of the hardware mandated number representation with the Common Lisp requirements of dynamic typing and garbage-collecting dynamic storage allocation.

Because of its type inference (section 7.3) and efficiency notes (section 7.11), Python is better than conventional Common Lisp compilers at ensuring that numerical expressions are statically typed. Python also goes somewhat farther than existing compilers in the area of allowing native machine number representations in the presence of garbage collection.

7.9.1. Descriptors

Common Lisp's dynamic typing requires that it be possible to represent any value with a fixed length object, known as a *descriptor*. This fixed-length requirement is implicit in features such as:

- Data types (like `simple-vector`) that can contain any type of object, and that can be destructively modified to contain different objects (of possibly different types.)
- Functions that can be called with any type of argument, and that can be redefined at run time.

In order to save space, a descriptor is invariably represented as a single word. Objects that can be directly represented in the descriptor itself are said to be *immediate*. Descriptors for objects larger than one word are in reality pointers to the memory actually containing the object.

Representing objects using pointers has two major disadvantages:

- The memory pointed to must be allocated on the heap, so it must eventually be freed by the garbage collector. Excessive heap allocation of objects (or "consing") is inefficient in several ways. See section 7.10.2.
- Representing an object in memory requires the compiler to emit additional instructions to read the actual value in from memory, and then to write the value back after operating on it.

The introduction of garbage collection makes things even worse, since the garbage collector must be able to determine whether a descriptor is an immediate object or a pointer. This requires that a few bits in each descriptor be dedicated to the garbage collector. The loss of a few bits doesn't seem like much, but it has a major efficiency

implication — objects whose natural machine representation is a full word (integers and single-floats) cannot have an immediate representation. So the compiler is forced to use an unnatural immediate representation (such as **fixnum**) or a natural pointer representation (with the attendant consing overhead.)

7.9.2. Non-Descriptor Representations

From the discussion above, we can see that the standard descriptor representation has many problems, the worst being number consing. Common Lisp compilers try to avoid these descriptor efficiency problems by using *non-descriptor* representations. A compiler that uses non-descriptor representations can compile this function so that it does no number consing:

```
(defun multby (vec n)
  (declare (type (simple-array single-float (*)) vec)
           (single-float n))
  (dotimes (i (length vec))
    (setf (aref vec i)
          (* n (aref vec i)))))
```

If a descriptor representation were used, each iteration of the loop might cons two floats and do three times as many memory references.

As its negative definition suggests, the range of possible non-descriptor representations is large. The performance improvement from non-descriptor representation depends upon both the number of types that have non-descriptor representations and the number of contexts in which the compiler is forced to use a descriptor representation.

Many Common Lisp compilers support non-descriptor representations for float types such as **single-float** and **double-float** (section 7.9.7.) Python adds support for full word integers (section 7.9.6), characters (section 7.9.10) and system-area pointers (unconstrained pointers, see section 8.2.1.) Many Common Lisp compilers support non-descriptor representations for variables (section 7.9.3) and array elements (section 7.9.8.) Python adds support for non-descriptor arguments and return values in local call (section 7.9.9).

7.9.3. Variables

In order to use a non-descriptor representation for a variable or expression intermediate value, the compiler must be able to prove that the value is always of a particular type having a non-descriptor representation. Type inference (section 7.3) often needs some help from user-supplied declarations. The best kind of type declaration is a variable type declaration placed at the binding point:

```
(let ((x (car l)))
  (declare (single-float x))
  ...)
```

Use of **the**, or of variable declarations not at the binding form is insufficient to allow non-descriptor representation of the variable — with these declarations it is not certain that all values of the variable are of the right type. It is sometimes useful to introduce a gratuitous binding that allows the compiler to change to a non-descriptor representation, like:

```
(etypecase x
  ((signed-byte 32)
   (let ((x x))
     (declare (type (signed-byte 32) x))
     ...))
  ...)
```

The declaration on the inner **x** is necessary here due to a phase ordering problem. Although the compiler will eventually prove that the outer **x** is a **(signed-byte 32)** within that **etypecase** branch, the inner **x** would have been optimized away by that time. Declaring the type makes let optimization more cautious.

Note that storing a value into a global (or **special**) variable always forces a descriptor representation.

Wherever possible, you should operate only on local variables, binding any referenced globals to local variables at the beginning of the function, and doing any global assignments at the end.

Efficiency notes signal use of inefficient representations, so programmer's needn't continuously worry about the details of representation selection (see section 7.11.3.)

7.9.4. Generic Arithmetic

In COMMON LISP, arithmetic operations are *generic.*[10] The + function can be passed **fixnums**, **bignums**, **ratios**, and various kinds of **floats** and **complexes**, in any combination. In addition to the inherent complexity of **bignum** and **ratio** operations, there is also a lot of overhead in just figuring out which operation to do and what contagion and canonicalization rules apply. The complexity of generic arithmetic is so great that it is inconceivable to open code it. Instead, the compiler does a function call to a generic arithmetic routine, consuming many instructions before the actual computation even starts.

This is ridiculous, since even Common Lisp programs do a lot of arithmetic, and the hardware is capable of doing operations on small integers and floats with a single instruction. To get acceptable efficiency, the compiler special-cases uses of generic arithmetic that are directly implemented in the hardware. In order to open code arithmetic, several constraints must be met:

- All the arguments must be known to be a good type of number.
- The result must be known to be a good type of number.
- Any intermediate values such as the result of **(+ a b)** in the call **(+ a b c)** must be known to be a good type of number.
- All the above numbers with good types must be of the *same* good type. Don't try to mix integers and floats or different float formats.

The "good types" are **(signed-byte 32)**, **(unsigned-byte 32)**, **single-float** and **double-float**. See sections 7.9.5, 7.9.6 and 7.9.7 for more discussion of good numeric types.

float is not a good type, since it might mean either **single-float** or **double-float**. **integer** is not a good type, since it might mean **bignum**. **rational** is not a good type, since it might mean **ratio**. Note however that these types are still useful in declarations, since type inference may be able to strengthen a weak declaration into a good one, when it would be at a loss if there was no declaration at all (see section 7.3). The **integer** and **unsigned-byte** (or non-negative integer) types are especially useful in this regard, since they can often be strengthened to a good integer type.

Arithmetic with **complex** numbers is inefficient in comparison to float and integer arithmetic. Complex numbers are always represented with a pointer descriptor (causing consing overhead), and complex arithmetic is always closed coded using the general generic arithmetic functions. But arithmetic with complex types such as:

```
(complex float)
(complex fixnum)
```

is still faster than **bignum** or **ratio** arithmetic, since the implementation is much simpler.

Note: don't use / to divide integers unless you want the overhead of rational arithmetic. Use **truncate** even when you know that the arguments divide evenly.

You don't need to remember all the rules for how to get open-coded arithmetic, since efficiency notes will tell you when and where there is a problem — see section 7.11.

---

[10] As Steele notes in CLTL II, this is a generic conception of generic, and is not to be confused with the CLOS concept of a generic function.

7.9.5. Fixnums

A fixnum is a "FIXed precision NUMber". In modern Common Lisp implementations, fixnums can be represented with an immediate descriptor, so operating on fixnums requires no consing or memory references. Clever choice of representations also allows some arithmetic operations to be done on fixnums using hardware supported word-integer instructions, somewhat reducing the speed penalty for using an unnatural integer representation.

It is useful to distinguish the **fixnum** type from the fixnum representation of integers. In Python, there is absolutely nothing magical about the **fixnum** type in comparison to other finite integer types. **fixnum** is equivalent to (is defined with **deftype** to be) **(signed-byte 30)**. **fixnum** is simply the largest subset of integers that *can be represented* using an immediate fixnum descriptor.

Unlike in other COMMON LISP compilers, it is in no way desirable to use the **fixnum** type in declarations in preference to more restrictive integer types such as **bit**, **(integer -43 7)** and **(unsigned-byte 8)**. Since Python does understand these integer types, it is preferable to use the more restrictive type, as it allows better type inference (see section 7.3.4.)

The small, efficient fixnum is contrasted with bignum, or "BIG NUMber". This is another descriptor representation for integers, but this time a pointer representation that allows for arbitrarily large integers. Bignum operations are less efficient than fixnum operations, both because of the consing and memory reference overheads of a pointer descriptor, and also because of the inherent complexity of extended precision arithmetic. While fixnum operations can often be done with a single instruction, bignum operations are so complex that they are always done using generic arithmetic.

A crucial point is that the compiler will use generic arithmetic if it can't *prove* that all the arguments, intermediate values, and results are fixnums. With bounded integer types such as **fixnum**, the result type proves to be especially problematical, since these types are not closed under common arithmetic operations such as +, -, * and /. For example, **(1+ (the fixnum x))** does not necessarily evaluate to a **fixnum**. Bignums were added to Common Lisp to get around this problem, but they really just transform the correctness problem "if this add overflows, you will get the wrong answer" to the efficiency problem "if this add *might* overflow then your program will run slowly (because of generic arithmetic.)"

There is just no getting around the fact that the hardware only directly supports short integers. To get the most efficient open coding, the compiler must be able to prove that the result is a good integer type. This is an argument in favor of using more restrictive integer types: **(1+ (the fixnum x))** may not always be a **fixnum**, but **(1+ (the (unsigned-byte 8) x))** always is. Of course, you can also assert the result type by putting in lots of **the** declarations and then compiling with **safety 0**.

7.9.6. Word Integers

Python is unique in its efficient implementation of arithmetic on full-word integers through non-descriptor representations and open coding. Arithmetic on any subtype of these types:

```
(signed-byte 32)
(unsigned-byte 32)
```

is reasonably efficient, although subtypes of **fixnum** remain somewhat more efficient.

If a word integer must be represented as a descriptor, then the **bignum** representation is used, with its associated consing overhead. The support for word integers in no way changes the language semantics, it just makes arithmetic on small bignums vastly more efficient. It is fine to do arithmetic operations with mixed **fixnum** and word integer operands; just declare the most specific integer type you can, and let the compiler decide what representation to use.

In fact, to most users, the greatest advantage of word integer arithmetic is that it effectively provides a few guard bits on the fixnum representation. If there are missing assertions on intermediate values in a fixnum expression, the intermediate results can usually be proved to fit in a word. After the whole expression is evaluated, there will often be a fixnum assertion on the final result, allowing creation of a fixnum result without even checking for overflow.

The remarks in section 7.9.5 about fixnum result type also apply to word integers; you must be careful to give the compiler enough information to prove that the result is still a word integer. This time, though, when we blow out of word integers we land in into generic bignum arithmetic, which is much worse than sleazing from **fixnums** to word integers. Note that mixing **(unsigned-byte 32)** arguments with arguments of any signed type (such as **fixnum**) is a no-no, since the result might not be unsigned.

7.9.7. Floats

Arithmetic on objects of type **single-float** and **double-float** is efficiently implemented using non-descriptor representations and open coding. As for integer arithmetic, the arguments must be known to be of the same float type. Unlike for integer arithmetic, the results and intermediate values usually take care of themselves due to the rules of float contagion, i.e. **(1+ (the single-float x))** is always a **single-float**.

Although they are not specially implemented, **short-float** and **long-float** are also acceptable in declarations, since they are synonyms for the **single-float** and **double-float** types, respectively. It is harmless to use list-style float type specifiers such as **(single-float 0.0 1.0)**, but Python currently makes little use of bounds on float types.

When a float must be represented as a descriptor, a pointer representation is used, creating consing overhead. For this reason, you should try to avoid situations (such as full call and non-specialized data structures) that force a descriptor representation. See sections 7.9.8 and 7.9.9.

See section 2.2 for information on the extensions to support IEEE floating point.

7.9.8. Specialized Arrays

COMMON LISP supports specialized array element types through the **:element-type** argument to **make-array**. When an array has a specialized element type, only elements of that type can be stored in the array. From this restriction comes two major efficiency advantages:

- A specialized array can save space by packing multiple elements into a single word. For example, a **base-char** array can have 4 elements per word, and a **bit** array can have 32. This space-efficient representation is possible because it is not necessary to separately indicate the type of each element.
- The elements in a specialized array can be given the same non-descriptor representation as the one used in registers and on the stack, eliminating the need for representation conversions when reading and writing array elements. For objects with pointer descriptor representations (such as floats and word integers) there is also a substantial consing reduction because it is not necessary to allocate a new object every time an array element is modified.

These are the specialized element types currently supported:

```
bit
(unsigned-byte 2)
(unsigned-byte 4)
(unsigned-byte 8)
(unsigned-byte 16)
(unsigned-byte 32)
base-character
single-float
double-float
```

Although a **simple-vector** can hold any type of object, **t** should still be considered a specialized array type, since arrays with element type **t** are specialized to hold descriptors.

When using non-descriptor representations, it is particularly important to make sure that array accesses are open-coded, since in addition to the generic operation overhead, efficiency is lost when the array element is converted to a descriptor so that it can be passed to (or from) the generic access routine. You can detect inefficient array accesses by enabling efficiency notes, see section 7.11. See also section 7.8.3.

7.9.9. Interactions With Local Call

Local call has many advantages (see section 7.6); one relevant to our discussion here is that local call extends the usefulness of non-descriptor representations. If the compiler knows from the argument type that an argument has a non-descriptor representation, then the argument will be passed in that representation. The easiest way to ensure that the argument type is known at compile time is to always declare the argument type in the called function, like:

```
(defun 2+f (x)
  (declare (single-float x))
  (+ x 2.0))
```

The advantages of passing arguments and return values in a non-descriptor representation are the same as for non-descriptor representations in general: reduced consing and memory access (see section 7.9.2.) This extends the applicative programming styles discussed in section 7.6 to numeric code. Also, if source files are kept reasonably small, block compilation can be used to reduce number consing to a minimum.

Note that non-descriptor return values can only be used with the known return convention (section 7.6.6.) If the compiler can't prove that a function always returns the same number of values, then it must use the unknown values return convention, which requires a descriptor representation. Pay attention to the known return efficiency notes to avoid number consing.

7.9.10. Characters

Python also uses a non-descriptor representation for characters when convenient. This improves the efficiency of string manipulation, but is otherwise pretty invisible; characters have an immediate descriptor representation, so there is not a great penalty for converting a character to a descriptor. Nonetheless, it may sometimes be helpful to declare character-valued variables as **base-character**.

7.10. General Efficiency Hints

This section is a summary of various implementation costs and ways to get around them. These hints are relatively unrelated to the use of the Python compiler, and probably also apply to most other Common Lisp implementations. In each section, there are references to related in-depth discussion.

7.10.1. Compile Your Code

At this point, the advantages of compiling code relative to running it interpreted probably need not be emphasized too much, but remember that in CMU Common Lisp, compiled code typically runs hundreds of times faster than interpreted code. Also, compiled (**fasl**) files load significantly faster than source files, so it is worthwhile compiling files which are loaded many times, even if the speed of the functions in the file is unimportant.

Even disregarding the efficiency advantages, compiled code is as good or better than interpreted code. Compiled code can be debugged at the source level (see chapter 5), and compiled code does more error checking. For these reasons, the interpreter should be regarded mainly as an interactive command interpreter, rather than as a programming language implementation.

Do not be concerned about the performance of your program until you see its speed compiled. Some techniques that make compiled code run faster make interpreted code run slower.

7.10.2. Avoid Unnecessary Consing

Consing is another name for allocation of storage, as done by the `cons` function (hence its name.) `cons` is by no means the only function which conses — so does `make-array` and many other functions. Arithmetic and function call can also have hidden consing overheads. Consing hurts performance in the following ways:

- Consing reduces memory access locality, increasing paging activity.
- Consing takes time just like anything else.
- Any space allocated eventually needs to be reclaimed, either by garbage collection or by starting a new "`lisp`" process.

Consing is not undiluted evil, since programs do things other than consing, and appropriate consing can speed up the real work. It would certainly save time to allocate a vector of intermediate results that are reused hundreds of times. Also, if it is necessary to copy a large data structure many times, it may be more efficient to update the data structure non-destructively; this somewhat increases update overhead, but makes copying trivial.

Note that the remarks in section 7.1.5 about the importance of separating tuning from coding also apply to consing overhead. The majority of consing will be done by a small portion of the program. The consing hot spots are even less predictable than the CPU hot spots, so don't waste time and create bugs by doing unnecessary consing optimization. During initial coding, avoid unnecessary side-effects and cons where it is convenient. If profiling reveals a consing problem, *then* go back and fix the hot spots.

See section 7.9.2 for a discussion of how to avoid number consing in Python.

7.10.3. Complex Argument Syntax

Common Lisp has very powerful argument passing mechanisms. Unfortunately, two of the most powerful mechanisms, rest arguments and keyword arguments, have a significant performance penalty:

- With keyword arguments, the called function has to parse the supplied keywords by iterating over them and checking them against the desired keywords.
- With rest arguments, the function must cons a list to hold the arguments. If a function is called many times or with many arguments, large amounts of memory will be allocated.

Although rest argument consing is worse than keyword parsing, neither problem is serious unless thousands of calls are made to such a function. The use of keyword arguments is strongly encouraged in functions with many arguments or with interfaces that are likely to be extended, and rest arguments are often natural in user interface functions.

Optional arguments have some efficiency advantage over keyword arguments, but their syntactic clumsiness and lack of extensibility has caused many COMMON LISP programmers to abandon use of optionals except in functions that have obviously simple and immutable interfaces (such as `subseq`), or in functions that are only called in a few places. When defining an interface function to be used by other programmers or users, use of only required and keyword arguments is recommended.

Parsing of `defmacro` keyword and rest arguments is done at compile time, so a macro can be used to provide a convenient syntax with an efficient implementation. If the macro-expanded form contains no keyword or rest arguments, then it is perfectly acceptable in inner loops.

Keyword argument parsing overhead can also be avoided by use of inline expansion (section 7.7) and block compilation (section 7.6.5.)

Note: the compiler open-codes most heavily used system functions which have keyword or rest arguments, so that no run-time overhead is involved.

7.10.4. Mapping and Iteration

One of the traditional Common Lisp programming styles is a highly applicative one, involving the use of mapping functions and many lists to store intermediate results. To compute the sum of the square-roots of a list of numbers, one might say:

```
(apply #'+ (mapcar #'sqrt list-of-numbers))
```

This programming style is clear and elegant, but unfortunately results in slow code. There are two reasons why:

- The creation of lists of intermediate results causes much consing (see 7.10.2).
- Each level of application requires another scan down the list. Thus, disregarding other effects, the above code would probably take twice as long as a straightforward iterative version.

An example of an iterative version of the same code:

```
(do ((num list-of-numbers (cdr num))
     (sum 0 (+ (sqrt (car num)) sum)))
    ((null num) sum))
```

See sections 7.3.1 and 7.4.1 for a discussion of the interactions of iteration constructs with type inference and variable optimization. Also, section 7.6.4 discusses an applicative style of iteration.

7.10.5. Trace Files and Disassembly

In order to write efficient code, you need to know the relative costs of different operations. The main reason why writing efficient Common Lisp code is difficult is that there are so many operations, and the costs of these operations vary in obscure context-dependent ways. Although efficiency notes point out some problem areas, the only way to ensure generation of the best code is to look at the assembly code output.

The **disassemble** function is a convenient way to get the assembly code for a function, but it can be very difficult to interpret, since the correspondence with the original source code is weak. A better (but more awkward) option is to use the **:trace-file** argument to **compile-file** to generate a trace file.

A trace file is a dump of the compiler's internal representations, including annotated assembly code. Each component in the program gets three pages in the trace file (separated by "^L"):

- The implicit-continuation (or IR1) representation of the optimized source. This is a dump of the flow graph representation used for "source level" optimizations. As you will quickly notice, it is not really very close to the source. This representation is not very useful to even sophisticated users.
- The Virtual Machine (VM, or IR2) representation of the program. This dump represents the generated code as sequences of "Virtual OPerations" (VOPs.) This representation is intermediate between the source and the assembly code — each VOP corresponds fairly directly to some primitive function or construct, but a given VOP also has a fairly predictable instruction sequence. An operation (such as +) may have multiple implementations with different cost and applicability. The choice of a particular VOP such as **+/fixnum** or **+/single-float** represents this choice of implementation. Once you are familiar with it, the VM representation is probably the most useful for determining what implementation has been used.
- An assembly listing, annotated with the VOP responsible for generating the instructions. This listing is useful for figuring out what a VOP does and how it is implemented in a particular context, but its large size makes it more difficult to read.

Note that trace file generation takes much space and time, since the trace file is tens of times larger than the

source file. To avoid huge confusing trace files and much wasted time, it is best to separate the critical program portion into its own file and then generate the trace file from this small file.

7.11. Efficiency Notes

Efficiency notes are messages that warn the user that the compiler has chosen a relatively inefficient implementation for some operation. Usually an efficiency note reflects the compiler's desire for more type information. If the type of the values concerned is known to the programmer, then additional declarations can be used to get a more efficient implementation.

Efficiency notes are controlled by the **extensions:inhibit-warnings** optimization quality (see section 6.7.1.) When **speed** is greater than **extensions:inhibit-warnings**, efficiency notes are enabled. Note that this implicitly enables efficiency notes whenever **speed** is increased from its default of **1**.

Consider this program with an obscure missing declaration:

```
(defun eff-note (x y z)
  (declare (fixnum x y z))
  (the fixnum (+ x y z)))
```

If compiled with **(speed 3) (safety 0)**, this note is given:

```
In: DEFUN EFF-NOTE
  (+ X Y Z)
==>
  (+ (+ X Y) Z)
Note: Forced to do inline (signed-byte 32) arithmetic (cost 3).
      Unable to do inline fixnum arithmetic (cost 2) because:
      The first argument is a (INTEGER -1073741824 1073741822),
      not a FIXNUM.
```

This efficiency note tells us that the result of the intermediate computation **(+ x y)** is not known to be a **fixnum**, so the addition of the intermediate sum to **z** must be done less efficiently. This can be fixed by changing the definition of **eff-note**:

```
(defun eff-note (x y z)
  (declare (fixnum x y z))
  (the fixnum (+ (the fixnum (+ x y)) z)))
```

7.11.1. Type Uncertainty

The main cause of inefficiency is the compiler's lack of adequate information about the types of function argument and result values. Many important operations (such as arithmetic) have an inefficient general (generic) case, but have efficient implementations that can usually be used if there is sufficient argument type information.

Type efficiency notes are given when a value's type is uncertain. There is an important distinction between values that are *not known* to be of a good type (uncertain) and values that are *known not* to be of a good type. Efficiency notes are given mainly for the first case (uncertain types.) If it is clear to the compiler that that there is not an efficient implementation for a particular function call, then an efficiency note will only be given if the **extensions:inhibit-warnings** optimization quality is **0** (see section 6.7.1.)

In other words, the default efficiency notes only suggest that you add declarations, not that you change the semantics of your program so that an efficient implementation will apply. For example, compilation of this form will not give an efficiency note:

```
(elt (the list l) i)
```

even though a vector access is more efficient than indexing a list.

7.11.2. Efficiency Notes and Type Checking

It is important that the `eff-note` example above used `(safety 0)`. When type checking is enabled, you may get apparently spurious efficiency notes. With `(safety 1)`, the note has this extra line on the end:

```
The result is a (INTEGER -1610612736 1610612733), not a FIXNUM.
```

This seems strange, since there is a `the` declaration on the result of that second addition.

In fact, the inefficiency is real, and is a consequence of Python's treating declarations as assertions to be verified. The compiler can't assume that the result type declaration is true — it must generate the result and then test whether it is of the appropriate type.

In practice, this means that when you are tuning a program to run without type checks, you should work from the efficiency notes generated by unsafe compilation. If you want code to run efficiently with type checking, then you should pay attention to all the efficiency notes that you get during safe compilation. Since user supplied output type assertions (e.g., from `the`) are disregarded when selecting operation implementations for safe code, you must somehow give the compiler information that allows it to prove that the result truly must be of a good type. In our example, it could be done by constraining the argument types more:

```
(defun eff-note (x y z)
  (declare (type (unsigned-byte 18) x y z))
  (+ x y z))
```

Of course, this declaration is acceptable only if the arguments to `eff-note` always *are* `(unsigned-byte 18)` integers.

7.11.3. Representation Efficiency Notes

When operating on values that have non-descriptor representations (see section 7.9.2), there can be a substantial time and consing penalty for converting to and from descriptor representations. For this reason, the compiler gives an efficiency note whenever it is forced to do a representation coercion more expensive than `*efficiency-note-cost-threshold*`.

Inefficient representation coercions may be due to type uncertainty, as in this example:

```
(defun set-flo (x)
  (declare (single-float x))
  (prog ((var 0.0))
    (setq var (gorp))
    (setq var x)
    (return var)))
```

which produces this efficiency note:

```
In: DEFUN SET-FLO
  (SETQ VAR X)
Note: Doing float to pointer coercion (cost 13) from X to VAR.
```

The variable `var` is not known to always hold values of type `single-float`, so a descriptor representation must be used for its value. In sort of situation, and adding a declaration will eliminate the inefficiency.

Often inefficient representation conversions are not due to type uncertainty — instead, they result evaluating a non-descriptor expression in a context that requires a descriptor result:

- Assignment to or initialization of any data structure other than a specialized array (see section 7.9.8), or
- Assignment to a `special` variable, or
- Passing as an argument or returning as a value in any function call that is not a local call (see section 7.9.9.)

If such inefficient coercions appear in a "hot spot" in the program, data structures redesign or program reorganization may be necessary to improve efficiency. See sections 7.6.5, 7.9 and 7.12.

Because representation selection is done rather late in compilation, the source context in these efficiency notes is somewhat vague, making interpretation more difficult. This is a fairly straightforward example:

```
(defun cf+ (x y)
  (declare (single-float x y))
  (cons (+ x y) t))
```

which gives this efficiency note:

```
In: DEFUN CF+
  (CONS (+ X Y) T)
Note: Doing float to pointer coercion (cost 13), for:
      The first argument of CONS.
```

The source context form is almost always the form that receives the value being coerced (as it is in the preceding example), but can also be the source form which generates the coerced value. Compiling this example:

```
(defun if-cf+ (x y)
  (declare (single-float x y))
  (cons (if (grue) (+ x y) (snoc)) t))
```

produces this note:

```
In: DEFUN IF-CF+
  (+ X Y)
Note: Doing float to pointer coercion (cost 13).
```

In either case, the note's text explanation attempts to include additional information about what locations are the source and destination of the coercion. Here are some example notes:

```
(IF (GRUE) X (SNOC))
Note: Doing float to pointer coercion (cost 13) from X.

  (SETQ VAR X)
Note: Doing float to pointer coercion (cost 13) from X to VAR.
```

Note that the return value of a function is also a place to which coercions may have to be done:

```
(DEFUN F+ (X Y) (DECLARE (SINGLE-FLOAT X Y)) (+ X Y))
Note: Doing float to pointer coercion (cost 13) to "<return value>".
```

Sometimes the compiler is unable to determine a name for the source or destination, in which case the source context is the only clue.

7.11.4. Verbosity Control

These variables control the verbosity of efficiency notes:

efficiency-note-cost-threshold [*Variable*]

Before printing some efficiency notes, the compiler compares the value of this variable to the difference in cost between the chosen implementation and the best potential implementation. If the difference is not greater than this limit, then no note is printed. The units are implementation dependent; the initial value suppresses notes about "trivial" inefficiencies. A value of 1 will note any inefficiency.

efficiency-note-limit [*Variable*]

When printing some efficiency notes, the compiler reports possible efficient implementations. The initial value of 2 prevents excessively long efficiency notes in the common case where there is no type information, so all implementations are possible.

7.12. Profiling

The first step in improving a program's performance is to profile the activity of the program to find where it spends its time. The best way to do this is to use the **profile** library entry (see section 3.6). Hemlock's "**Lisp Library**" command will help you find, describe, and load this tool. Basically, it provides a macro **profile** for encapsulating routines for statistics gathering and **report-time** that prints for each named function the following information:

- The total CPU time used in the function for all calls to it.
- The total number of bytes consed in the function for all calls to it.
- The total number of calls.
- The average amount of CPU time per call.

The following routines are in the released system:

See **time** (page 7). This macro executes a form and reports the time to do so and how much it consed. Due to the nature of the interpreter, you can get more accurate consing information if you put **time** in a function, compile it, and then call the function. For things which execute fairly quickly, time them more than once, since there may be more paging overhead in the first timing. Also, you can put what you are really interested in an iteration form and time that; then divide the time and bytes consed by the number of iterations for more accurate statistics.

extensions:get-bytes-consed [*Function*]

This function returns the number of bytes allocated since the first time you called it. The first time it is called it returns zero. The above profiling routines use this to report consing information.

7.12.1. A Note on Timing

There are two general kinds of timing information provided by the **time** macro and other profiling utilities: real time and run time. Real time is elapsed, wall clock time. It will be affected in a fairly obvious way by any other activity on the machine. The more other processes contending for CPU and memory, the more real time will increase. This means that real time measurements are difficult to replicate, though this is less true on a dedicated workstation. The advantage of real time is that it is real. It tells you really how long the program took to run under the benchmarking conditions. The problem is that you don't know exactly what those conditions were.

Run time is the amount of time that the processor supposedly spent running the program, as opposed to waiting for I/O or running other processes. "User run time" and "system run time" are numbers reported by the Unix kernel. They are supposed to be a measure of how much time the processor spent running your "user" program (which will include GC overhead, etc.), and the amount of time that the kernel spent running "on your behalf".

Ideally, user time should be totally unaffected by benchmarking conditions; in reality user time does depend on other system activity, though in rather non-obvious ways.

System time will clearly depend on benchmarking conditions. In Lisp benchmarking, paging activity increases system run time (but not by as much as it increases real time, since the kernel spends some time waiting for the disk, and this is not run time, kernel or otherwise.)

In my experience, the biggest trap in interpreting kernel/user run time is to look only at user time. In reality, it seems that the *sum* of kernel and user time is more reproducible. The problem is that as system activity increases, there is a spurious *decrease* in user run time. In effect, as paging, etc., increases, user time leaks into system time.

So, in practice, the only way to get truly reproducible results is to run with the same competing activity on the

system. Try to run on a machine with nobody else logged in, and check with "ps aux" to see if there are any system processes munching large amounts of CPU or memory. If the ratio between real time and the sum of user and system time varies much between runs, then you have a problem.

7.12.2. Benchmarking Techniques

Given these imperfect timing tools, how do should you do benchmarking? The answer depends on whether you are trying to measure improvements in the performance of a single program on the same hardware, or if you are trying to compare the performance of different programs and/or different hardware.

For the first use (measuring the effect of program modifications with constant hardware), you should look at *both* system+user and real time to understand what effect the change had on CPU use, and on I/O (including paging.) If you are working on a CPU intensive program, the change in system+user time will give you a moderately reproducible measure of performance across a fairly wide range of system conditions. For a CPU intensive program, you can think of system+user as "how long it would have taken to run if I had my own machine." So in the case of comparing CPU intensive programs, system+user time is relatively real, and reasonable to use.

For programs that spend a substantial amount of their time paging, you really can't predict elapsed time under a given operating condition without benchmarking in that condition. User or system+user time may be fairly reproducible, but it is also relatively meaningless, since in a paging or I/O intensive program, the program is spending its time waiting, not running, and system time and user time are both measures of run time. A change that reduces run time might increase real time by increasing paging.

Another common use for benchmarking is comparing the performance of the same program on different hardware. You want to know which machine to run your program on. For comparing different machines (operating systems, etc.), the only way to compare that makes sense is to set up the machines in *exactly* the way that they will *normally* be run, and then measure *real* time. If the program will normally be run along with X, then run X. If the program will normally be run on a dedicated workstation, then be sure nobody else is on the benchmarking machine. If the program will normally be run on a machine with three other Lisp jobs, then run three other Lisp jobs. If the program will normally be run on a machine with 8meg of memory, then run with 8meg. Here, "normal" means "normal for that machine". If you the choice of an unloaded RT or a heavily loaded PMAX, do your benchmarking on an unloaded RT and a heavily loaded PMAX.

If you have a program you believe to be CPU intensive, then you might be tempted to compare "run" times across systems, hoping to get a meaningful result even if the benchmarking isn't done under the expected running condition. Don't to this, for two reasons:

- The operating systems might not compute run time in the same way.
- Under the real running condition, the program might not be CPU intensive after all.

In the end, only real time means anything — it is the amount of time you have to wait for the result. The only valid uses for run time are:

- To develop insight into the program. For example, if run time is much less than elapsed time, then you are probably spending lots of time paging.
- To evaluate the relative performance of CPU intensive programs in the same environment.

Chapter 8

MACH Interface

By Rob Maclachlan, Skef Wholey,
Bill Chiles, and William Lott

CMU Common Lisp attempts to make the full power of the underlying environment available to the Lisp programmer. This is done using combination of hand-coded interfaces, automatically generated MACH RPC stubs and foreign function calls to C libraries. Although the techniques differ, the style of interface is similar. This chapter provides an overview of the facilities available and general rules for using them, as well as describing specific features in detail. It is assumed that the reader has a working familiarity with Mach, Unix and X, as well as access to the standard system documentation.

8.1. Lisp Equivalents for C Routines

The MACH documentation describes the system interface in terms of C procedure headers. The corresponding Lisp function will have a somewhat different interface, since Lisp argument passing conventions and datatypes are different.

The main difference in the argument passing conventions is that Lisp does not support passing values by reference. In Lisp, all argument and results are passed by value. Interface functions take some fixed number of arguments and return some fixed number of values. A given "parameter" in the C specification will appear as an argument, return value, or both, depending on whether it is an In parameter, Out parameter, or In/Out parameter. The basic transformation one makes to come up with the Lisp equivalent of a C routine is to remove the Out parameters from the call, and treat them as extra return values. In/Out parameters appear both as arguments and return values. Since Out and In/Out parameters are only conventions in C, you must determine the usage from the documentation.

Thus, the C routine declared as

```
kern_return_t lookup(servport, portsname, portsid)
        port    servport;
        char    *portsname;
        int     *portsid;       /* out */
{
 ...
 *portsid = <expression to compute portsid field>
 return(KERN_SUCCESS);
}
```

has as its Lisp equivalent something like

```
(defun lookup (ServPort PortsName)
  ...
  (values
   success
   <expression to compute portsid field>))
```

An extra twist that complicates this "translation" process but makes programming easier is this: when the routine returns a record value, the components of that record may be returned as multiple values. This eliminates the need to extract fields from Alien structures (see below) and frees the programmer from having to explicitly deallocate such structures.

So, the C routine declared as

```
void getevent(servport, event)
        port    servport;
        keyevent *event;        /* out */

{
  ...
  keyevent->cmd = <expression to compute Cmd field>
  keyevent->ch = <expression to compute Ch field>
  keyevent->region = <expression to compute Region field>
  keyevent->y = <expression to compute Y field>
  keyevent->x = <expression to compute X field>
  ...
}
```

would be written like this in Lisp:

```
(defun getevent (servport)
  ...
  (values
   <expression to compute Cmd field>
   <expression to compute Ch field>
   <expression to compute Region field>
   <expression to compute Y field>
   <expression to compute X field>))
```

Fortunately, CMU Common Lisp programmers rarely have to worry about the nuances of this translation process, since the names of the arguments and return values are documented in a way so that the **describe** function (and the Hemlock **Describe Function Call** command, invoked with **C-M-Shift-A**) will list this information. Since the names of arguments and return values are usually descriptive, the information that **describe** prints is usually all one needs to write a call to a Matchmaker-generated function. Most programmers use this on-line documentation nearly all of the time, and thereby avoid the need to handle bulky manuals and perform the translation from barbarous tongues.

8.2. Type Translations

Lisp data types have very different representations from those used by conventional languages such as C. Since the system interfaces are designed for conventional languages, Lisp must translate objects to and from the Lisp representations. Many simple objects have a direct translation: integers, characters, strings and floating point numbers are translated to the corresponding Lisp object. A number of types, however, are implemented differently in Lisp for reasons of clarity and efficiency.

Instances of enumerated types are expressed as keywords in Lisp. Thus, an instance of the enumerated type defined by

```
Type KeyHowWait = (KeyWaitDiffPos, KeyDontWait, KeyWaitEvent);
```

would be written in Lisp as a keyword: `:keywaitdiffpos`, `:keydontwait`, or `:keywaitevent`.

Records, arrays, and pointer types are implemented with the Alien facility (see page 139.) Access functions are defined for these types which convert fields of records, elements of arrays, or data referenced by pointers into Lisp objects (possibly another object to be referenced with another access function):

- A record of type *type* can be constructed with a function `make-`*type*. A field named *field* of a record of type *type* may be accessed with a function *type-field*, and set with `setf` of that function.
- An array of type *type* can be constructed with a function `make-`*type*; if the array type allows for a variable upper bound on indices, these bounds may be specified. Elements of such an array may be accessed with the function *type*`-ref`, and may be set with `setf` of that function.
- A pointer of type *type* to an object may be dereferenced with a function `indirect-`*type*. To create an object and get a pointer of type *type* to that object, one can call the function `make-`*type*. If the pointer type references an array of objects, indices may be provided as optional arguments to the indirect function, and if the array has a variable upper bound, it may be specified when calling the constructor function.

One should dispose of Alien objects created by constructor functions or returned from remote procedure calls when they are no longer of any use, freeing the virtual memory associated with that object. Since Aliens contain pointers to non-Lisp data, the garbage collector cannot do this itself. If the Alien was created using MACH memory allocation (e.g. `vm_allocate`), then the storage should be freed using `dispose-alien` (page 140). If the memory was obtained from a foreign function call to a routine that used `malloc`, then `system:free` should be used on the `system:alien-sap` of the Alien.

8.2.1. System Area Pointers

Note that in some cases an address is represented by a Lisp integer, and in other cases it is represented by a real pointer. Pointers are usually used when an object in the current address space is being referred to. The MACH virtual memory manipulation calls must use integers, since in principle the address could be in any process, and Lisp cannot abide random pointers. Because these types are represented differently in Lisp, one must explicitly coerce between these representations.

System Area Pointers (SAPs) provide a mechanism that bypasses the Alien type system and accesses virtual memory directly. A SAP is a raw byte pointer into the `lisp` process address space. SAPs are represented with a pointer descriptor, so SAP creation can cause consing. However, the compiler uses a non-descriptor representation for SAPs when possible, so the consing overhead is generally minimal. See section 7.9.2.

`system:alien-sap` *alien* [*Macro*]

The function `alien-sap` is used to generate a system area pointer (a virtual address that points into the section of Lisp's address space reserved for Alien objects) from an Alien.

NOTE: Usually a pointer from a system interface function is an Alien, but whenever a pointer is passed in, it must be passed as a system area pointer. This strange calling convention was adopted to eliminate the necessity of constructing an Alien value just for the purpose of passing a pointer. The programmer interface is simplified, since a simple function call can be made in most places without the need to declare a local variable, construct an Alien value, and deallocate that Alien value (or use `alien-bind` for those three things).

system:sap-int *sap* [*Function*]
system:int-sap *int* [*Function*]
The function **sap-int** is used to generate an integer corresponding to the system area pointer, suitable for passing to the kernel interfaces (which want all addresses specified as integers). The function **int-sap** is used to do the opposite conversion. The integer representation of a SAP is the byte offset of the SAP from the start of the address space.

system:sap+ *sap offset* [*Function*]
This function adds a byte *offset* to *sap*, returning a new SAP.

system:sap-ref-8 *sap offset* [*Function*]
system:sap-ref-16 *sap offset* [*Function*]
system:sap-ref-32 *sap offset* [*Function*]
These functions return the 8, 16 or 32 bit unsigned integer at *offset* from *sap*. The *offset* is scaled by the integer size, so the offset for **sap-ref-16** is in 16 half-words, and the offset for **sap-ref-32** is in 32 bit words. **Setf** may be used with the these functions to deposit values into virtual memory.

system:signed-sap-ref-8 *sap offset* [*Function*]
system:signed-sap-ref-16 *sap offset* [*Function*]
system:signed-sap-ref-32 *sap offset* [*Function*]
These functions are the same as the above unsigned operations, except that they sign-extend, returning a negative number if the high bit is set.

8.3. Unix System Calls

You probably won't have much cause to use them, but all the Unix system calls are available. The Unix system call functions are in the "**mach**" package. The name of the interface for a particular system call is the name of the system call prepended with "**unix-**". The system usually defines the associated constants without any prefix name. To find out how to use a particular system call, try using **describe** on it. If that is unhelpful, look at the source in syscall.lisp or consult your system maintainer.

The Unix system calls indicate an error by returning **nil** as the first value and the Unix error number as the second value. If the call succeeds, then the first value will always be non-**nil**, often **t**.

mach:get-unix-error-msg *error* [*Function*]
This function returns a string describing the Unix error number *error*.

Many of the UNIX system calls return file descriptors. Instead of using other UNIX system calls to perform I/O on them, you can create a stream around them. For this purpose, fd-streams exist.

system:make-fd-stream *descriptor* **&key :input :output :element-type :buffering :name :file :original :delete-original** [*Function*]
This function creates a file descriptor stream using *descriptor*. If *input* is non-**nil**, input operations are allowed. If *output* is non-**nil**, output operations are allowed. The default is input only.

element-type is the type of the unit of transaction for the stream, which defaults to **string-char**. See the Common Lisp description of **open** for valid values.

buffering is the kind of output buffering desired for the stream. Legal values are **:none** for no buffering,

`:line` for buffering up to each newline, and `:full` for full buffering.

name is a simple-string name to use for descriptive purposes when the system prints an fd-stream. When printing fd-streams, the system prepends the streams name with `"Stream for "`. If *name* is unspecified, it defaults to a string containing *file* or *descriptor*, in order of preference.

If the created stream is a file stream, *file* is the name of the associated file, and this must be a simple-string. *original* is simple-string name of a backup file containing the original contents *file* while writing *file*.

When you abort the stream by passing `t` to `close` as the second argument, if you supplied both *file* and *original*, `close` will rename the *original* name to the *file* name.

When you `close` the stream normally, if you supplied *original*, and *delete-original* is non-`nil`, `close` deletes *original*.

`system:fd-stream-p` *object* [*Function*]

This function returns `t` if *object* is an fd-stream, and `nil` if not.

`system:fd-stream-fd` *stream* [*Function*]

This returns the file descriptor associated with *stream*.

8.4. Making Sense of Return Codes

Whenever a remote procedure call returns a Mach error code (such as `kern_return_t`), it is usually prudent to check that code to see if the call was successful. To relieve the programmer of the hassle of testing this value himself, and to centralize the information about the meaning of non-success return codes, CMU Common Lisp provides a number of macros and functions.

`system:gr-error` *function gr* `&optional` *context* [*Function*]

Signals a Lisp error, printing a message indicating that the call to the specified *function* failed, with the return code *gr*. If supplied, the *context* string is printed after the *function* name and before the string associated with the *gr*. For example:

```
* (gr-error 'nukegarbage 3 "lost big")

Error in function GR-ERROR:
NUKEGARBAGE lost big, no space.
Proceed cases:
0: Return to Top-Level.
Debug  (type H for help)
(Signal #<Conditions:Simple-Error.5FDE0>)
0]
```

`system:gr-call` *function* `&rest` *args* [*Macro*]

`system:gr-call*` *function* `&rest` *args* [*Macro*]

These macros can be used to call a function and automatically check the GeneralReturn code and signal an appropriate error in case of non-successful return. `gr-call` returns `nil` if no error occurs, while `gr-call*` returns the second value of the function called.

```
* (gr-call mach:port_allocate *task-self*)
NIL
*
```

system:gr-bind ({*var*}*) (*function* {*arg*}*) {*form*}* [*Macro*]

This macro can be used much like **multiple-value-bind** to bind the *vars* to return values resulting from calling the *function* with the given *args*. The first return value is not bound to a variable, but is checked as a GeneralReturn code, as in **gr-call**.

```
* (gr-bind (port_list port_list_cnt)
           (mach:port_select *task-self*)
    (format t "The port count is ~S." port_list_cnt)
    port_list)
The port count is 0.
#<Alien value>
*
```

8.5. Packages

The functions and constants that make up each Matchmaker-generated interface usually reside in their own package, and the public symbols of that package are exported. Thus, one usually uses the package for an interface one wishes to use. A program that used the Mach kernel, the CLX interface to the X window manager, and the Message Name server might begin with:

```
;;; -*- Package: Hack -*-
;;;
;;; A silly graphics hack.
;;; Written by Joe Schmoe.
;;;
(in-package "HACK" :use '("LISP" "XLIB" "MACH" "MSGN"))
```

Note that all of the standard interfaces are built into the CMU Common Lisp core image, and one doesn't need to load any other files to use these facilities. Here is a list of the packages that hold the built-in interfaces.

MACH Holds the MACH interface and the Unix system calls.

XLIB Holds the CLX interface to the X window manager version 11. See the CLX documentation for details.

8.6. Useful Variables

The information passed to the process in its startup message is available in the values of global variables.

system:*nameserverport* [*Variable*]

Port to the message name server.

system:*task-self* [*Variable*]
system:*task-data* [*Variable*]
system:*task-notify* [*Variable*]

The initial ports for the Lisp process.

8.7. Reading the Command Line

The shell parses the command line with which Lisp is invoked, and passes a data structure containing the parsed information to Lisp. This information is then extracted from that data structure and put into a set of Lisp data structures.

extensions:*command-line-strings* [*Variable*]
extensions:*command-line-utility-name* [*Variable*]
extensions:*command-line-words* [*Variable*]
extensions:*command-line-switches* [*Variable*]

The value of ***command-line-words*** is a list of strings that make up the command line, one word per string. The first word on the command line, i.e. the name of the program invoked (usually **"lisp"**) is stored in ***command-line-utility-name***. The value of ***command-line-switches*** is a list of **command-line-switch** structures, with a structure for each word on the command line starting with a hyphen. All the command line words between the program name and the first switch are stored in ***command-line-words***.

The following functions may be used to examine **command-line-switch** structures.

extensions:cmd-switch-name *switch* [*Function*]

Returns the name of the switch, less the preceding hyphen and trailing equal sign (if any).

extensions:cmd-switch-value *switch* [*Function*]

Returns the value designated using an embedded equal sign, if any. If the switch has no equal sign, then this is null.

extensions:cmd-switch-words *switch* [*Function*]

Returns a list of the words between this switch and the next switch or the end of the command line.

Chapter 9

Event Dispatching with SYSTEM:SERVE-EVENT

By Bill Chiles and Rob Maclachlan

It is common to have multiple activities simultaneously operating in the same Lisp process. Furthermore, Lisp programmers tend to expect a flexible development environment. It must be possible to load and modify application programs without requiring modifications to other running programs. CMU Common Lisp achieves this by having a central scheduling mechanism based on an event-driven, object-oriented paradigm.

An *event* is some interesting happening that should cause the Lisp process to wake up and do something. These events include X events and activity on Unix file descriptors. The object-oriented mechanism is only available with the first two, and it is optional with X events as described later in this chapter. In an X event, the window ID is the object capability and the X event type is the operation code. The Unix file descriptor input mechanism simply consists of an association list of a handler to call when input shows up on a particular file descriptor.

9.1. Object Sets

An *object set* is a collection of objects that have the same implementation for each operation. Externally the object is represented by the object capability and the operation is represented by the operation code. Within Lisp, the object is represented by an arbitrary Lisp object, and the implementation for the operation is represented by an arbitrary Lisp function. The object set mechanism maintains this translation from the external to the internal representation.

system:make-object-set *name* **&optional** *default-handler* [*Function*]

This function makes a new object set. *Name* is a string used only for purposes of identifying the object set when it is printed. *Default-handler* is the function used as a handler when an undefined operation occurs on an object in the set. You can define operations with the **serve-***operation* functions exported the **"EXTENSIONS"** package for X events (see section 9.4). Objects are added with **system:add-xwindow-object**. Initially the object set has no objects and no defined operations.

system:object-set-operation *object-set operation-code* [*Function*]

This function returns the handler function that is the implementation of the operation corresponding to *operation-code* in *object-set*. When set with **setf**, the setter function establishes the new handler. The **serve-***operation* functions exported from the **"EXTENSIONS"** package for X events (see section 9.4) call this on behalf of the user when announcing a new operation for an object set.

system:add-xwindow-object *window object object-set* [*Function*]

These functions add *port* or *window* to *object-set*. *Object* is an arbitrary Lisp object that is associated with the *port* or *window* capability. *Window* is a CLX window. When an event occurs, **system:serve-event** passes *object* as an argument to the handler function.

9.2. The SYSTEM:SERVE-EVENT Function

The **system:serve-event** function is the standard way for an application to wait for something to happen. For example, the Lisp system calls **system:serve-event** when it wants input from X or a terminal stream. The idea behind **system:serve-event** is that it knows the appropriate action to take when any interesting event happens. If an application calls **system:serve-event** when it is idle, then any other applications with pending events can run. This allows several applications to run "at the same time" without interference, even though there is only one thread of control. Note that if an application is waiting for input of any kind, then other applications will get events.

system:serve-event &optional *timeout* [*Function*]

This function waits for an event to happen and then dispatches to the correct handler function. If specified, *timeout* is the number of seconds to wait before timing out. A time out of zero seconds is legal and causes **system:serve-event** to poll for any events immediately available for processing. **system:serve-event** returns **t** if it serviced at least one event, and **nil** otherwise. Depending on the application, when **system:serve-event** returns **t**, you might want to call it repeatedly with a timeout of zero until it returns **nil**.

If input is available on any designated file descriptor, then this calls the appropriate handler function supplied by **system:add-fd-handler**.

Since events for many different applications may arrive simultaneously, an application waiting for a specific event must loop on **system:serve-event** until the desired event happens. Since programs such as Hemlock call **system:serve-event** for input, applications such as Matchmaker servers usually do not need to call **system:serve-event** at all; Hemlock allows other application's handlers to run when it goes into an input wait.

system:serve-all-events &optional *timeout* [*Function*]

This function is similar to **system:serve-event**, except it serves all the pending events rather than just one. It returns **t** if it serviced at least one event, and **nil** otherwise.

9.3. Using SYSTEM:SERVE-EVENT with Unix File Descriptors

Object sets are not available for use with file descriptors, as there are only two operations possible on file descriptors: input and output. Instead, a handler for either input or output can be registered with **system:serve-event** for a specific file descriptor. Whenever any input shows up, or output is possible on this file descriptor, the function associated with the handler for that descriptor is funcalled with the descriptor as it's single argument.

system:add-fd-handler *fd direction function* [*Function*]

This function installs and returns a new handler for the file descriptor *fd*. *Direction* can be either **:input** if the system should invoke the handler when input is available or **:output** if the system should invoke the handler when output is possible. This returns a unique object representing the handler, and this is a suitable argument for **system:remove-fd-handler** *Function* must take one argument,

the file descriptor.

system:remove-fd-handler *handler* [*Function*]
This function removes *handler*, that **add-fd-handler** must have previously returned.

system:with-fd-handler (*direction fd function*) {*form*}* [*Macro*]
This macro executes the supplied forms with a handler installed using *fd*, *direction*, and *function*. See **system:add-fd-handler**.

system:wait-until-fd-usable *direction fd* **&optional** *timeout* [*Function*]
This function waits for up to *timeout* seconds for *fd* to become usable for *direction* (either **:input** or **:output**). If *timeout* is **nil** or unspecified, this waits forever.

system:invalidate-descriptor *fd* [*Function*]
This function removes all handlers associated with *fd*. This should only be used in drastic cases (such as I/O errors, but not necessarily EOF). Normally, you should use **remove-fd-handler** to remove the specific handler.

9.4. Using SYSTEM:SERVE-EVENT with the CLX Interface to X

Remember from section 9.1, an object set is a collection of objects, CLX windows in this case, with some set of operations, event keywords, with corresponding implementations, the same handler functions. Since X allows multiple display connections from a given process, you can avoid using object sets if every window in an application or display connection behaves the same. If a particular X application on a single display connection has windows that want to handle certain events differently, then using object sets is a convenient way to organize this since you need some way to map the window/event combination to the appropriate functionality.

The following is a discussion of functions exported from the **"EXTENSIONS"** package that facilitate handling CLX events through **system:serve-event**. The first two routines are useful regardless of whether you use **system:serve-event**:

ext:open-clx-display &optional *string* [*Function*]
This function parses *string* for an X display specification including display and screen numbers. *String* defaults to the following:

```
(cdr (assoc :display ext:*environment-list* :test #'eq))
```

If any field in the display specification is missing, this signals an error. **ext:open-clx-display** returns the CLX display and screen.

ext:flush-display-events *display* [*Function*]
This function flushes all the events in *display*'s event queue including the current event, in case the user calls this from within an event handler.

9.4.1. Without Object Sets

Since most applications that use CLX, can avoid the complexity of object sets, these routines are described in a separate section. The routines described in the next section that use the object set mechanism are based on these interfaces.

ext:enable-clx-event-handling *display handler* [*Function*]
This function causes **system:serve-event** to notice when there is input on *display*'s connection to the X11 server. When this happens, **system:serve-event** invokes *handler* on *display* in a dynamic context with an error handler bound that flushes all events from *display* and returns. By returning, the error handler declines to handle the error, but it will have cleared all events; thus, entering the debugger will not result in infinite errors due to streams that wait via **system:serve-event** for input. Calling this repeatedly on the same *display* establishes *handler* as a new handler, replacing any previous one for *display*.

ext:disable-clx-event-handling *display* [*Function*]
This function undoes the effect of **ext:enable-clx-event-handling**.

ext:with-clx-event-handling (*display handler*) {**form**}* [*Macro*]
This macro evaluates each *form* in a context where **system:serve-event** invokes *handler* on *display* whenever there is input on *display*'s connection to the X server. This destroys any previously established handler for *display*.

9.4.2. With Object Sets

This section discusses the use of object sets and **system:serve-event** to handle CLX events. This is necessary when a single X application has distinct windows that want to handle the same events in different ways. Basically, you need some way of asking for a given window which way you want to handle some event because this event is handled differently depending on the window. Object sets provide this feature.

For each CLX event-key symbol-name *XXX* (for example, *key-press*), there is a function **serve-***XXX* of two arguments, an object set and a function. The **serve-***XXX* function establishes the function as the handler for the **:***XXX* event in the object set. Recall from section 9.1, **system:add-xwindow-object** associates some Lisp object with a CLX window in an object set. When **system:serve-event** notices activity on a window, it calls the function given to **ext:enable-clx-event-handling**. If this function is **ext:object-set-event-handler**, it calls the function given to **serve-***XXX*, passing the object given to **system:add-xwindow-object** and the event's slots as well as a couple other arguments described below.

To use object sets in this way:

1. Create an object set.
2. Define some operations on it using the **serve-***XXX* functions.
3. Add an object for every window on which you receive requests. This can be the CLX window itself or some structure more meaningful to your application.
4. Call **system:serve-event** to service an X event.

ext:object-set-event-handler *display* [*Function*]
This function is a suitable argument to **ext:enable-clx-event-handling**. The actual event handlers defined for particular events within a given object set must take an argument for every slot in the appropriate event. In addition to the event slots, **ext:object-set-event-handler** passes the following arguments:

- The object, as established by **system:add-xwindow-object**, on which the event occurred.
- event-key, see **xlib:event-case**.
- send-event-p, see **xlib:event-case**.

Describing any **ext:serve-***event-key-name* function, where *event-key-name* is an event-key symbol-

name (for example, **ext:serve-key-press**), indicates exactly what all the arguments are in their correct order.

When creating an object set for use with **ext:object-set-event-handler**, specify **ext:default-clx-event-handler** as the default handler for events in that object set. If no default handler is specified, and the system invokes the default default handler, it will cause an error since this function takes arguments suitable for handling port messages.

9.5. A SYSTEM:SERVE-EVENT Example

This section contains two examples using **system:serve-event**. The first one does not use object sets, and the second, slightly more complicated one does.

9.5.1. Without Object Sets

This example defines an input handler for a CLX display connection. It only recognizes **:key-press** events. The body of the example loops over **system:serve-event** to get input.

```
(in-package "SERVER-EXAMPLE")

(defun my-input-handler (display)
  (xlib:event-case (display :timeout 0)
    (:key-press (event-window code state)
     (format t "KEY-PRESSED (Window = ~D) = ~S.~%"
             (xlib:window-id event-window)
             ;; See Hemlock Command Implementor's Manual for convenient
             ;; input mapping function.
             (ext:translate-character display code state))
     ;; Make XLIB:EVENT-CASE discard the event.
     t)))
```

```
(defun server-example ()
  "An example of using the SYSTEM:SERVE-EVENT function and object sets to
   handle CLX events."
  (let* ((display (ext:open-clx-display))
         (screen (display-default-screen display))
         (black (screen-black-pixel screen))
         (white (screen-white-pixel screen))
         (window (create-window :parent (screen-root screen)
                                :x 0 :y 0 :width 200 :height 200
                                :background white :border black
                                :border-width 2
                                :event-mask
                                (xlib:make-event-mask :key-press))))
    ;; Wrap code in UNWIND-PROTECT, so we clean up after ourselves.
    (unwind-protect
        (progn
          ;; Enable event handling on the display.
          (ext:enable-clx-event-handling display #'my-input-handler)
          ;; Map the windows to the screen.
          (map-window window)
          ;; Make sure we send all our requests.
          (display-force-output display)
          ;; Call serve-event for 100,000 events or immediate timeouts.
          (dotimes (i 100000) (system:serve-event)))
      ;; Disable event handling on this display.
      (ext:disable-clx-event-handling display)
      ;; Get rid of the window.
      (destroy-window window)
      ;; Pick off any events the X server has already queued for our
      ;; windows, so we don't choke since SYSTEM:SERVE-EVENT is no longer
      ;; prepared to handle events for us.
      (loop
       (unless (deleting-window-drop-event *display* window)
        (return)))
      ;; Close the display.
      (xlib:close-display display))))

(defun deleting-window-drop-event (display win)
  "Check for any events on win.  If there is one, remove it from the
   event queue and return t; otherwise, return nil."
  (xlib:display-finish-output display)
  (let ((result nil))
    (xlib:process-event
     display :timeout 0
     :handler #'(lambda (&key event-window &allow-other-keys)
                  (if (eq event-window win)
                      (setf result t)
                      nil)))
    result))
```

9.5.2. With Object Sets

This example involves more work, but you get a little more for your effort. It defines two objects, **input-box** and **slider**, and establishes a **:key-press** handler for each object, **key-pressed** and **slider-pressed**. We have two object sets because we handle events on the windows manifesting these objects differently, but the events come over the same display connection.

```
(in-package "SERVER-EXAMPLE")

(defstruct (input-box (:print-function print-input-box)
                      (:constructor make-input-box (display window)))
  "Our program knows about input-boxes, and it doesn't care how they
   are inplemented."
  display       ; The CLX display on which my input-box is displayed.
  window)       ; The CLX window in which the user types.
;;;
(defun print-input-box (object stream n)
  (declare (ignore n))
  (format stream "#<Input-Box ~S>" (input-box-display object)))

(defvar *input-box-windows*
        (system:make-object-set "Input Box Windows"
                                #'ext:default-clx-event-handler))

(defun key-pressed (input-box event-key event-window root child
                    same-screen-p x y root-x root-y modifiers time
                    key-code send-event-p)
  "This is our :key-press event handler."
  (declare (ignore event-key root child same-screen-p x y
                   root-x root-y time send-event-p))
  (format t "KEY-PRESSED (Window = ~D) = ~S.~%"
          (xlib:window-id event-window)
          ;; See Hemlock Command Implementor's Manual for convenient
          ;; input mapping function.
          (ext:translate-character (input-box-display input-box)
                                   key-code modifiers)))
;;;
(ext:serve-key-press *input-box-windows* #'key-pressed)
```

```
(defstruct (slider (:print-function print-slider)
                   (:include input-box)
                   (:constructor %make-slider
                                 (display window window-width max)))
  "Our program knows about sliders too, and these provide input values
   zero to max."
  bits-per-value  ; bits per discrete value up to max.
  max)            ; End value for slider.
;;;
(defun print-slider (object stream n)
  (declare (ignore n))
  (format stream "#<Slider ~S  0..~D>"
          (input-box-display object)
          (1- (slider-max object))))
;;;
(defun make-slider (display window max)
  (%make-slider display window
                (truncate (xlib:drawable-width window) max)
                max))

(defvar *slider-windows*
        (system:make-object-set "Slider Windows"
                                #'ext:default-clx-event-handler))

(defun slider-pressed (slider event-key event-window root child
                       same-screen-p x y root-x root-y modifiers time
                       key-code send-event-p)
  "This is our :key-press event handler for sliders.  Probably this is
   a mouse thing, but for simplicity here we take a character typed."
  (declare (ignore event-key root child same-screen-p x y
                   root-x root-y time send-event-p))
  (format t "KEY-PRESSED (Window = ~D) = ~S  -->  ~D.~%"
          (xlib:window-id event-window)
          ;; See Hemlock Command Implementor's Manual for convenient
          ;; input mapping function.
          (ext:translate-character (input-box-display slider)
                                   key-code modifiers)
          (truncate x (slider-bits-per-value slider))))
;;;
(ext:serve-key-press *slider-windows* #'slider-pressed)
```

```
(defun server-example ()
  "An example of using the SYSTEM:SERVE-EVENT function and object sets to
   handle CLX events."
  (let* ((display (ext:open-clx-display))
         (screen (display-default-screen display))
         (black (screen-black-pixel screen))
         (white (screen-white-pixel screen))
         (iwindow (create-window :parent (screen-root screen)
                                 :x 0 :y 0 :width 200 :height 200
                                 :background white :border black
                                 :border-width 2
                                 :event-mask
                                 (xlib:make-event-mask :key-press)))
         (swindow (create-window :parent (screen-root screen)
                                 :x 0 :y 300 :width 200 :height 50
                                 :background white :border black
                                 :border-width 2
                                 :event-mask
                                 (xlib:make-event-mask :key-press)))
         (input-box (make-input-box display iwindow))
         (slider (make-slider display swindow 15)))
    ;; Wrap code in UNWIND-PROTECT, so we clean up after ourselves.
    (unwind-protect
        (progn
          ;; Enable event handling on the display.
          (ext:enable-clx-event-handling display
                                         #'ext:object-set-event-handler)
          ;; Add the windows to the appropriate object sets.
          (system:add-xwindow-object iwindow input-box
                                     *input-box-windows*)
          (system:add-xwindow-object swindow slider
                                     *slider-windows*)
          ;; Map the windows to the screen.
          (map-window iwindow)
          (map-window swindow)
          ;; Make sure we send all our requests.
          (display-force-output display)
          ;; Call server for 100,000 events or immediate timeouts.
          (dotimes (i 100000) (system:serve-event)))
      ;; Disable event handling on this display.
      (ext:disable-clx-event-handling display)
      (delete-window iwindow display)
      (delete-window swindow display)
      ;; Close the display.
      (xlib:close-display display))))
```

```
(defun delete-window (window display)
  ;; Remove the windows from the object sets before destroying them.
  (system:remove-xwindow-object window)
  ;; Destroy the window.
  (destroy-window window)
  ;; Pick off any events the X server has already queued for our
  ;; windows, so we don't choke since SYSTEM:SERVE-EVENT is no longer
  ;; prepared to handle events for us.
  (loop
   (unless (deleting-window-drop-event display window)
     (return))))

(defun deleting-window-drop-event (display win)
  "Check for any events on win.  If there is one, remove it from the
   event queue and return t; otherwise, return nil."
  (xlib:display-finish-output display)
  (let ((result nil))
    (xlib:process-event
     display :timeout 0
     :handler #'(lambda (&key event-window &allow-other-keys)
                  (if (eq event-window win)
                      (setf result t)
                      nil)))
    result))
```

Chapter 10

The Alien Facility

By Rob Maclachlan

10.1. What the Alien Facility Is

Aliens provide a mechanism in Lisp for manipulating objects which are foreign to the Lisp environment. Aliens are used in the foreign function calling interface, matchmaker interfaces to the Mach specific system calls and the name server, and to call Unix system calls. The Alien functions and macros described in this chapter allow Lisp objects to be converted from the Lisp representation to other representations as expected in C code or IPC messages and vice versa.

10.2. Alien Values

Objects in messages are manipulated via typed pointers to the data involved. These typed pointers are called *Alien values*. An Alien value is a Lisp object which consists of three components:

address — The address of the object pointed to. This is a word address, which may in general be a ratio, since objects need not be word aligned.

size — The size in bits of the object pointed to. This information is used to make sure that accesses to the object fall within it.

type — The Alien type of the object pointed to. Since Alien values have a type, functions that use them can check that their arguments are of the correct type.

10.3. Alien Types

Alien types are tags attached to Alien values that may be checked to assure that they are not used inappropriately. When types are compared the comparison is done with the Lisp **equal** function. Types are typically represented by symbols or lists of symbols such as the following:

```
string
(directory-entry type-file)
(signed-byte 7)
string-char
```

A convention which is encouraged, but not enforced, is that an ordinary type is represented by a symbol, and a type with some subtype information, such as a discriminated union is represented as a list of the main type and the

subtype information.

10.4. Alien Primitives

This section describes the defined Alien primitives. Some of these primitives are intended to be used only in code generated by matchmaker, while others might be used by mere mortals.

system:make-alien *type size* **&optional** *address* [*Function*]
Make an Alien object of type *type* that is *size* bits long. *address* may be either a number, **:static** or **:dynamic**. If address is a number, then that becomes the returned alien's address. If *address* is **:static** or **:dynamic** then storage is allocated to hold the data. Aliens that are allocated statically are packed as many as will fit on a page, resulting in increased storage efficiency, but disallowing the deallocation of the storage. Since static aliens are allocated contiguously, the **save** function can arrange to save their contents, permitting initialization of such Aliens to be done only once. Dynamic Aliens are allocated on page boundaries, and may be deallocated using **dispose-alien**.

system:alien-type *alien* [*Function*]
system:alien-size *alien* [*Function*]
system:alien-address *alien* [*Function*]
These functions return the type, size and address of *alien*, respectively.

system:alien-sap *alien* [*Function*]
This function returns the address of *alien* as a system-area-pointer. If the address is not an integer, an error will be signaled, since it cannot be represented as a system-area-pointer.

system:copy-alien *alien* [*Function*]
Copy the storage pointed to by *alien* and return a new Alien value that describes it.

system:alien-assign *to-alien from-alien* [*Function*]
Copies the bits in *from-alien* into *to-alien*. The alien values must be of the same size and type.

system:dispose-alien *alien* [*Function*]
Release any storage associated with *alien*. Any reference to *alien* afterward may lose horribly.

system:alien-access *alien* **&optional** *lisp-type* [*Function*]
alien-access returns the object described by *alien* as a Lisp object of type *lisp-type*. An error is signalled if the type of *alien* cannot be converted to the given *lisp-type*. For most lisp-types the corresponding Alien type is identical. If the Lisp type is uniquely determined by the type of the *alien* then *lisp-type* need not be supplied.

lisp-type must be one of the following types:

(unsigned-byte *n***)**
An unsigned integer *n* bits wide, as in Common Lisp.

(signed-byte *n***)**
A signed integer *n* bits wide.

boolean A one bit value, represented in Lisp as **t** or **nil**.

(system:enumeration *name***)**
Access a value of the enumeration *name*. Enumerations are defined by the macro **defenumeration** (page 141).

string-char An eight-bit ASCII character.

simple-string The corresponding Alien type is **system:perq-string** which is a Perq Pascal string (a string whose first byte is a count of the remaining characters). A second Alien type **system:null-terminated-string** has been defined which allows passing and receiving C style strings.

system:port A Mach IPC port.

single-float double-float
There are two alien types one for **single-float** and one for **double-float**.

system:system-area-pointer
Return as a system-area-pointer the long-word described by *alien*. It is an error for the address not to be in the system area. This lisp type may also be used with the **alien** alien type.

If **alien-access** is set with **setf** then the inverse type conversion is done, and the alien set to the new value. When setting, additional types are available:

(system:pointer *type*)
type may be any unboxed Lisp type such as **simple-string**, **simple-bit-vector** and **(simple-array (unsigned-byte 8))**. When an object of such a type is stored the address of the first data word is stored in the corresponding location.

(system:alien *type* [*size*])
This lisp type is used to access a pointer as an alien value. When read, an alien value created out of the pointer, *type* and *size* is returned. When set, the address of the alien values is written. When read, the *size* must be specified, when set it is ignored.

system:defenumeration *name* {{*element*}$^+$ | {(*element value*)}$^+$}* [*Macro*]
Define an enumeration type for use with **alien-access**. The enumeration may be used with the **enumeration** Alien type by specifying its *name*. Each successive *element* is assigned a numeric value, starting at zero. Each element must be a keyword symbol. Example:

```
(defenumeration era :stone-age :medieval :now :space-age)

(setf (alien-access (language-era (alien-value pascal))
                    (enumeration era))
      :stone-age)
```

The numeric value for an element may be specified by using a list of the keyword and the numeric value. If the value is specified for any element then it must be specified for all. Each value must be an integer.

```
(defenumeration silly (:a -32) (:b 15) (:c 1000000))
```

10.5. Alien Variables

An Alien variable is a symbol that has an Alien value associated with it. An Alien variable is not a Lisp variable -- in order to obtain the value of an Alien variable, the special form **alien-value** must be used. The reason for using Alien variables as opposed to Lisp variables is that various additional information can be associated with the Alien variable which may permit code which refers to it to be compiled more efficiently.

system:alien-value *name* [*Special form*]
Return the value of the Alien variable *name*.

system:alien-bind ({ (*name value type* [*aligned*]) }*) {*form*}* [*Special form*]
Alien-bind defines a local Alien variable *name* having the specified Alien *value*. Bindings are done serially, as by **let***. If *aligned* is supplied and is non-nil, then the *value* is asserted to be word aligned. Hopefully this feature will be replaced with something less silly.

system:defalien *name type size* [*address*] [*Macro*]
Defines *name* as an Alien variable, creating a value from *type*, *size* and *address* as for **make-alien** (page 140). *Name* and *type* are not evaluated. Since the alien-value for a defalien created variable is kept in the value cell of the symbol it is not necessary (but legal) to use **alien-value** to obtain the value.

10.6. Alien Stacks

For some purposes it is useful to have stack allocation of Alien values. Alien stacks are used by Matchmaker to receive messages into, since a software interrupt may cause an interface to be entered recursively.

system:define-alien-stack *name type size* [*Macro*]
Defines a stack of static Aliens having the specified *type* and *size*. The stack has no maximum size, since new Aliens are allocated whenever they are needed.

system:with-stack-alien (*var name*) {*form*}* [*Special form*]
Binds the Alien variable *var* to an Alien value from the Alien stack with the specified *name* during the evaluation of the *forms*.

10.7. Alien Operators

An Alien operator is a function which returns an Alien value. When an Alien operator is defined via the **defoperator** macro, the type of the result and all of the Alien valued arguments is specified. If an argument to an Alien operator is not the of the correct type an error is signalled. Because of the way an Alien operator is specified, it can be compiled much more efficiently than a function that does the same thing.

system:defoperator (*name result-type*) ({ (*arg arg-type*) | *arg*}*) [*doc-string*] *body* [*Macro*]
This macro defines *name* as an Alien operator returning a value of type *result-type*. *Doc-string*, if supplied, becomes the function documentation for the function created.

The *args* to the operator are similar to the binding specifiers to **alien-bind** (page 142). If the type of the argument is specified, then the argument must be an Alien value of the specified type, otherwise it may be any Lisp value.

Defoperator is similar to the complex form of **defsetf** or **defmacro** in that the body is evaluated at compile time, the result of the evaluation being the desired code. When the body is evaluated, Lisp variables having the arguments' names are bound to markers which must appear in the resulting code where a reference to that argument is desired. Normally the form which results from the evaluation of the body consists solely of combinations of **alien-index** and **alien-indirect** on arguments and simple numeric functions thereof.

system:alien-index *alien offset size* [*Function*]
This function indexes into *alien* by *offset* bits and returns an Alien value *size* bits long. It is an error for the field so selected not to fit inside *alien*. Normally this function is used only within the definition of an Alien operator, so the type of the resulting value is **nil** to indicate that it has no particular type

system:alien-indirect *alien size* [*Function*]

This function takes a word at the place described by *alien* and treats them as a pointer, returning a new Alien value which describes the piece of memory pointed to by that pointer which is *size* bits long. It is an error for *alien* not to describe a piece of storage suitable for use as a pointer. Like **alien-index**, this is normally only used within the definition of an Alien operator, and its result type is **nil**.

system:long-words *n* [*Function*]
system:words *n* [*Function*]
system:bytes *n* [*Function*]
system:bits *n* [*Function*]

These functions are equivalent to multiplication by thirty-two, sixteen, eight and one respectively. They also assert their argument to be an integer. Use of these function in the definition of Alien operators can make the definition clearer, and give additional information that can be used to produce better compiled code.

10.8. Examples

This C declaration might be translated into the following Alien operator definitions:

```
struct foo {
    int a;
    struct foo *b[100];
};

struct foo f;

 <==>

;;; This operator selects the A field from a Foo.  The type of the
;;; resulting Alien is (signed-byte 32), which is what a C int is.
;;; It takes one argument called Foo which is an Alien value of type
;;;  Foo. Since A is the first field in the record, we index into
;;; the Alien by zero bits.  The size of the result is thirty-two bits,
;;; or one  long-word.  Alien-Value must be used on the parameter,
;;; since it is an Alien variable.
;;;
(defoperator (foo-a (signed-byte 16)) ((foo foo))
  `(alien-index (alien-value ,foo) 0 (long-words 1)))

;;; This operator extracts the B field from a Foo.  The result type is
;;; (ref (array (ref foo) 100)), indicating that it is a pointer to an
;;; array of pointers to foos.  Note the use of list Alien types to
;;; indicate subtype information, but remember that this is merely a
;;; convention.  The B field is one long-word into the record, and since
;;; it is a pointer, it is thirty-two bits, or one long-word long.
;;;
(defoperator (foo-b (ref (array (ref foo) 100))) ((foo foo))
  `(alien-index (alien-value ,foo) (long-words 1) (long-words 1)))

;;; This operator dereferences a pointer to an (array (ref foo) 100).  The
;;; size of the resulting Alien is one hundred long-words, since the array
;;; contains one hundred thirty-two bit pointers
;;;
(defoperator (deref-array-ref-foo-100 (array (ref foo) 100))
```

```
            ((ra (ref (array (ref foo) 100))))
  `(alien-indirect (alien-value ,ra) (long-words 100)))

;;; Index into an (array (ref foo) 100).  Here we have a non-alien-valued
;;; parameter I, which is the index into the array.
;;;
(defoperator (index-array-ref-foo-100 (ref foo))
            ((a (array (ref foo) 100)) i)
  `(alien-index (alien-value ,a) (long-words ,i) (long-words 1)))

;;; Dereference a pointer to a foo.  A foo is two long-words.
;;;
(defoperator (deref-foo foo) ((rfoo (ref foo)))
  `(alien-indirect (alien-value ,rfoo) (long-words 2)))

;;; Define F as an Alien variable, whose type is foo and is three words
;;; long.  Storage to hold the foo will be allocated.
;;;
(defalien f foo (long-words 2))
```

With this definition, the following C expression could be translated in this way:

```
f.b[7].a

 <==>

(alien-access
 (foo-a (deref-foo (index-array-ref-foo-100
                    (deref-array-ref-foo-100 (foo-b (alien-value f)))
                    7))))
```

If instead of getting the A out of the seventh foo, we wanted a vector containing the first F.A foos in the array F.B, we could do this:

```
;; Find how many foos to use by getting the A field.
(let* ((num (alien-access (foo-a (alien-value f))))
       (result (make-array num)))
  ;;
  ;; Bind the Alien value for the array so we don't have to keep
  ;; recomputing it.
  (alien-bind ((a (deref-array-ref-foo-100 (foo-b (alien-value f))))
               (array (ref foo) 100))
    ;;
    ;; Loop over the first N elements and stash them in the result vector.
    (dotimes (i num)
      (setf (svref result i)
            (deref-foo (index-array-ref-foo-100 (alien-value a) i))))
    result))
```

Chapter 11

Foreign Function Call Interface

By David B. McDonald

11.1. Introduction

The foreign function call interface allows a Lisp program to call functions written in other languages. The current implementation of the foreign function call interface assumes a C calling convention and thus routines written in any language that adheres to this convention may be called from Lisp. Several functions and macros are made available to load object files into the currently running Lisp, to define data structures to be passed to or received from foreign routines, and to define the interface to a foreign function.

The foreign function call interface relies heavily on the primitives provided by the alien facility. If you intend to use the full power of the foreign function call interface, you will need to become familiar with the facilities provided by aliens. See the previous chapter for details.

Lisp sets up various interrupt handling routines and other environment information when it first starts up and expects these to be in place at all times. The C functions called by Lisp should either not change the environment, especially the interrupt entry points, or should make sure that these entry points are restored when the C function returns to Lisp. If a C function makes changes without restoring things to the way they were when the C function was entered, there is no telling what will happen.

11.2. Loading Unix Object Files

There is a single function that loads in one or more Unix object files into the currently running Lisp.

`extensions:load-foreign` *files* **`&optional`** *libraries linker base-file env* [*Function*]

Load-foreign loads a list of Unix object files into the currently running Lisp. *Files* should be a simple-string specifying the name of a single Unix object file or a list of such strings. *Libraries* should be a list of simple-strings specifying libraries in a format that **ld**, the Unix linker, expects. The default value for *libraries* is '("-lc") (i.e., the standard C library). *Linker* should specify the Unix linker to use when linking the object files. The default is "/usr/cs/bin/ld". *Base-file* is the file to use for the initial symbol table information. The default is the Lisp start up code ("/usr/misc/.cmucl/bin/lisp"). *Env* should be a list of simple strings in the format of Unix environment variables (i.e., "A=B", where A is an environment variable and B is its value).The default value for *env* is the environment information available at the time Lisp was invoked. Unless you are certain that you want to change this, you should just use the default.

Load-foreign runs a Unix linker (default "/usr/cs/bin/ld") on the files and libraries (in the order given to

load-foreign) creating an absolute Unix object file. This object file is then loaded into a memory at the correct location. All the external symbols that define either routines or variables are placed in a hash table for use by the macros that define interfaces to foreign routines. Note that load-foreign must be run before the any references to foreign functions or variables are made.

11.3. Defining Foreign Data Types

There are several data types that are pre-defined and can be used directly for defining interfaces to routines. There are also facilities for defining more complicated data structures such as arrays, structures, and pointers.

The following table gives a list of the pre-defined data types and the corresponding Lisp data type provided by the foreign function interface:

C Data Type	Lisp Data Type
int or long	(signed-byte 32)
unsigned int or long	(unsigned-byte 32)
short	(signed-byte 16)
unsigned short	(unsigned-byte 16)
char	(signed-byte 8)
unsigned char	(unsigned-byte 8)
float	short-float
double	long-float
procedure pointer	system:c-procedure

If you need to know how many bits are being used to represent a particular data structure, you can use the following function.

extensions:c-sizeof *c-type* [*Function*]

C-sizeof accepts a C type specification and returns the number of bits needed to represent it. For example, (**c-sizeof** 'int) returns 32.

11.3.1. Defining New C Types

extensions:def-c-type *name spec* [*Macro*]

Def-C-Type defines the symbol *name* to be a C type as specified by *spec*. *Spec* can either be a previously defined C type, or an alien type such as (signed-byte 32) or (system:null-terminated-string 256). This mechanism provides a short hand for referring to a particular type in other definitions.

For example, int above is defined by the following call to def-c-type:

```
(def-c-type int (signed-byte 32))
```

11.3.2. Defining C Arrays

extensions:def-c-array *name element-type* **&optional** *size* [*Macro*]

Def-C-array defines a C array type with name *name*. *Element-type* specifies the type of each element of the array. The optional parameter *size* specifies the number of elements in the array.

Def-C-array creates the following functions and forms that can be used to manipulate a C array:

make-*name* This function is used to allocate an array. Note that def-c-array does not actually create any storage for the array. You must use this routine to do that. If the *size*

parameter is specified in the call to def-c-array, then make-*name* accepts no arguments and returns an alien value of the appropriate size. Otherwise, it accepts one argument which should be the number of elements desired for this particular instantiation of the array. In either case, an alien value is returned and can be used to refer to the storage for the array.

name-ref — This setfable form allows you to refer to a particular element of an array. It accepts two arguments an alien value such as returned by make-*name* and an index. It picks up the correct element out of the array and returns it as the value. You can use setf on this form to set an element of an array.

For example, it is possible define an array type, create an instance of it, and set the first element of the newly created instance with the following code:

```
(def-c-array arr int 10)
(setq x (make-arr))
(setf (alien-access (arr-ref x 0)) 10)
```

11.3.3. Defining C Records

extensions:def-c-record *name* { (*sname stype*) }* [*Macro*]

Def-c-record defines a C record. This macro actually defines two C types *Name* is the name of the record and **Name* is the name of the pointer to the record. This is useful for record structures that have pointers to themselves as one or more of the slots. Following the *name* of the record are a list of (*sname stype*) pairs. These are the name and the type of a slot, respectively. As with def-c-array, def-c-record does not allocate any storage to hold data. It just defines the type. It also defines the function make-*name* which can be used to create an instance of the record. This will allocate storage to hold the record and return an alien value that refers to that particular record. For each field in the record, a setfable operator (named *name-sname*) is created, so that it is possible to reference and set particular fields of a record.

As an example, the following C structure definition and lisp def-c-record define equivalent data structures:

```
struct c-struct {
        short x, y;
        char a, b;
        int z;
        c-struct *n;
};

(def-c-record c-struct
        (x short)
        (y short)
        (a char)
        (b char)
        (z int)
        (n *c-struct))
```

To create an instance of c-struct and assign values to fields, the following code could be used:

```
(setq cs (make-c-struct))

(setf (alien-access (c-struct-x cs)) 20)
(setf (alien-access (c-struct-a cs)) 5)
(setf (alien-access (c-struct-n cs) 'alien) cs)
```

11.3.4. Defining C Pointers

C allows one to have pointers to other C-types. This can be done using the def-c-pointer macro as follows:

extensions:def-c-pointer *name to* [*Macro*]

Def-c-pointer defines *name* to be a C type that is a pointer to the C type specified by to.

For example, it is possible to define a pointer to an int by the the following:

```
(def-c-pointer *int int)
```

11.4. Defining Variable Interfaces

It is sometimes necessary to be able to refer to a global C variable. The macro def-c-variable allows this.

extensions:def-c-variable *name type* [*Macro*]

Def-c-variable makes global C variables accessible from Lisp. *Name* should be a simple-string with the exact capitalization of the C variable to which you want to be able to refer (C is case sensitive and so must be the name provided). This macro creates a Lisp symbol with *name* (uppercased) whose value is an alien value with type *type* that can be used to access the global C variable.

For example, it is often necessary to read the global C variable errno to determine why a particular function call failed. It is possible to define errno and make it accessible from Lisp by the following:

```
(def-c-variable "errno" int)
```

Now it is possible to get the value of the C variable errno by doing the following:

```
(alien-access errno)
```

11.5. Defining Routine Interfaces

There is a single macro that defines the interface to a C function. Note that all the types that it uses must be defined before you define the interface, otherwise errors will occur.

extensions:def-c-routine *name rtype* **&rest** *spec* [*Macro*]

Def-c-routine defines a Lisp function that interfaces to a C routine. *Name* should be a simple string with the exact capitalization of the C function (since C is case sensitive) or a list of two elements. The first element should be a simple-string as above and the second should be a symbol which is used as the Lisp name of the function. If this second form is not used, a symbol with *name* uppercased is used as the name of the Lisp function.

Rtype is the type of the return value and should be one of the builtin C-types or a user defined one. The special type extensions:void can be used if the C routine returns no useful value as its standard return value. Currently, double floats can not be returned by C functions. If the function returns a pointer and the result coming back is C NULL (0), then the function will return NIL. Also, if the result is a C String, then a Lisp string is returned instead of the alien value pointing to the C string.

Spec is bound to a list of the rest of the forms in the call to def-c-routine. Each element of this list should have the following form:

(*aname atype* [*amode*] {**options**}*)

Where *aname* should be a symbol and is used as the name of the argument. *Atype* should be a symbol that is associated with a C type. If you are passing floating point numbers to a C routine, you should

declare the type of the parameter as a long-float or double. This is because C passes all floating point parameters as double floats. The routine may be called with any type of number, since it will be coerced to a long-float before being passed on to C. *Options* is currently ignored. *Amode* should be one of the following:

:in	This specifies that the argument is passed by value. This is the default. No value for this argument is returned by the Lisp function when this mode is used.
:out	The type of the argument must be a pointer to a fixed sized object (such as a record or fixed size array). An object of the correct size is allocated and passed to the C routine by reference. When the C routine finishes, the contents of this object are returned as one of the values to the calling function. If the object returned is a record or array, it will be copied to a new alien value which will be returned.
:copy	This is similar to :in, but the argument is copied to a pre-allocated object and a pointer to this object is passed to the C routine.
:in-out	A combination of :copy and :out. The argument is copied to a pre-allocated object and a pointer to this object is passed to the C routine. On return, a new alien value is allocated for the object and returned as a multiple value.

For example, the C function cfoo with the following calling conventions:

```
cfoo (a, i)
    char a;
    int i;
{
/* Body of cfoo. */
}
```

can be described by the following call to def-c-routine:

```
(def-c-routine ("cfoo" lfoo) (void)
        (a char)
        (i int))
```

11.6. Calling Lisp routines from C

There is currently a mechanism for calling Common Lisp functions from C, but it is rather restricted, and is scheduled for replacement. If you need to call Common Lisp functions from C, contact us and we will let you know what capabilities are available in the system you have.

11.7. An Example

This section presents a complete example of an interface to a somewhat complicated C function. This example should give a fairly good idea of how to get the effect you want for almost any kind of C function.

Suppose you have the following C function which you want to be able to call from Lisp in the file cfun.c:

```
struct cfunr {
    int x;
    char *s;
};

struct cfunr *cfun (i, s, r, a)
    int i;
    char *s;
    struct cfunr *r;
    int a[10];
{   int j;
    struct cfunr *r2;

    printf("i = %d\n", i);
    printf("s = %s\n", s);
    printf("r->x = %d\n", r->x);
    printf("r->s = %s\n", r->s);
    for (j = 0; j < 10; j++) printf("a[%d] = %d.\n", j, a[j]);
    r2 = (struct cfunr *) malloc (sizeof(struct cfunr));
    r2->x = i + 5;
    r2->s = "A C string";
    return(r2);
};
```

It is possible to call this function from Lisp using the file cfun.lisp whose contents is:

```
;;; -*- Package: test-c-call; Mode: Lisp -*-
(in-package "TEST-C-CALL" :use '("LISP" "SYSTEM" "EXTENSIONS"))

;;; Define c-string as a null-terminated string of up to 256 characters.
(def-c-type c-string (null-terminated-string 256))

;;; Define a *c-string to be a pointer to a c-string.
(def-c-pointer *c-string c-string)

;;; Define the record cfunr in Lisp.
(def-c-record cfunr
        (x int)
        (s *c-string))

;;; Define the C array ar to have 10 elements of type int.
(def-c-array ar int 10)

;;; Define the C type pointer to the array above.
(def-c-pointer *ar ar)
```

```
;;; Load in the C object file with the function definition.
(load-foreign "cfun.o")

;;; Define the Lisp function interface to the C routine.  It returns a
;;; pointer to a record of type cfunr.  It accepts four parameters: i,
;;; an int; s, a pointer to a string; r, a pointer to a cfunr record;
;;; and a, a pointer to the array defined above.
(def-c-routine "cfun" (*cfunr)
        (i int)
        (s *c-string)
        (r *cfunr)
        (a *ar))

;;; A function which sets up the parameters to the C function and
;;; actually calls it.
(defun call-cfun ()
  (let ((arr (make-ar))                    ; Make an array.
        (rec (make-cfunr)))                ; Make a record.
    (alien-bind ((a arr ar t)
                 (r rec cfunr t))
      (dotimes (i 10)                      ; Fill array.
        (setf (alien-access (ar-ref (alien-value a) i)) i))
      (setf (alien-access (cfunr-x (alien-value r))) 20)
      (setf (alien-access (cfunr-s (alien-value r)) 'pointer)
            "A Lisp String")
      (let ((rec2 (cfun 5 "Another Lisp String"
                        (alien-sap (alien-value r))
                        (alien-sap (alien-value a)))))
        (format t "Returned from C function.~%")
        (alien-bind ((r2 rec2 cfunr t))
          (let ((cs (alien-access (cfunr-s (alien-value r2)) 'alien)))
            (alien-bind ((s cs (null-terminated-string 256) t))
              (values (alien-access (cfunr-x (alien-value r2)))
                      (alien-access (alien-value s)))))))))))
```

To execute the above example, it is necessary to compile the c routine as follows:

```
cc -c cfun.c
```

Once this has been done, you should start up lisp, and do the following:

```
lisp
;;; Lisp should start up with its normal prompt.

;;; Next compile the lisp file
* (compile-file "cfun.lisp")
Error output from cfun.lisp 17-Mar-87 17:09:57.
Compiled on 18-Mar-87 17:33:16 by CLC version M1.6 (16-Mar-87).

INDIRECT-*C-STRING compiled.
MAKE-CFUNR compiled.
INDIRECT-*CFUNR compiled.
CFUNR-X compiled.
CFUNR-S compiled.
MAKE-AR compiled.
AR-REF compiled.
INDIRECT-*AR compiled.
CFUN compiled.
Warning in CALL-CFUN:
  Could not show 32 bit store to be word-aligned:
(AR-REF A I)
CALL-CFUN compiled.

Finished compilation of file "/usr/dbm/cfun.lisp".
0 Errors, 1 Warnings.
Elapsed time 0:00:10, run time 0:00:09.

;;; Now load the file:
* (load "cfun")
;;; Lisp prints out the following information:
[Loading foreign files (cfun.o) ...
  [Running ld ... done.]
  [Reading Unix object file ... done.]
  [Loading symbol table information ... done.]
done.]
T
```

```
;;; Now call the routine that sets up the parameters and calls the C
;;; function.
* (test-c-call::call-cfun)
;;; The C routine prints the following information to standard output.
i = 5
s = Another Lisp string
r->x = 20
r->s = A Lisp string
a[0] = 0.
a[1] = 1.
a[2] = 2.
a[3] = 3.
a[4] = 4.
a[5] = 5.
a[6] = 6.
a[7] = 7.
a[8] = 8.
a[9] = 9.
;;; Lisp prints out the following information.
Returned from C function.
;;; Return values from the call to test-c-call::call-cfun.
10
"A C string"
*
```

If any of the foreign functions do output, they should not be called from within Hemlock. Depending on the situation, various strange behavior occurs. On the console, you will see no output; under X, the output goes to the window in which Lisp was started; on a terminal, the output will be placed in the current buffer but will not be recognized by Hemlock. This means it will overwrite information already in the window and be overwritten by Hemlock. This will not have any impact on the contents of the buffer, since the output is coming from a source that Hemlock does not know about.

Chapter 12

Interprocess Communication under LISP

Written by William Lott and Bill Chiles

CMU Common Lisp offers a facility for interprocess communication (IPC) on top of using Unix system calls and the complications of that level of IPC. There is a simple remote-procedure-call (RPC) package build on top of TCP/IP sockets.

12.1. The REMOTE Package

The `"REMOTE"` package provides simple RPC facility including interfaces for creating servers, connecting to already existing servers, and calling functions in other Lisp processes. The routines for establishing a connection between two processes, `create-request-server` and `connect-to-remote-server`, return *wire* structures. A wire maintains the current state of a connection, and all the RPC forms require a wire to indicate where to send requests.

12.1.1. Connecting Servers and Clients

Before a client can connect to a server, it must know the network address on which the server accepts connections. Network addresses consist of a host address or name, and a port number. Host addresses are either a string of the form `"VANCOUVER.SLISP.CS.CMU.EDU"` or a 32 bit unsigned integer. Port numbers are 16 bit unsigned integers. Note: *port* in this context has nothing to do with Mach ports and message passing.

When a process wants to receive connection requests (that is, become a server), it first picks an integer to use as the port. Only one server (Lisp or otherwise) can use a given port number on a given machine at any particular time. This can be an iterative process to find a free port: picking an integer and calling `create-request-server`. This function signals an error if the chosen port is unusable. You will probably want to write a loop using `handler-case`, catching conditions of type error, since this function does not signal more specific conditions.

`wire:create-request-server` *port* `&optional` *on-connect* [*Function*]

`create-request-server` sets up the current Lisp to accept connections on the given port. If port is unavailable for any reason, this signals an error. When a client connects to this port, the acceptance mechanism makes a wire structure and invokes the *on-connect* function. Invoking this function has a couple purposes, and *on-connect* may be `nil` in which case the system foregoes invoking any function at connect time.

The *on-connect* function is both a hook that allows you access to the wire created by the acceptance mechanism, and it confirms the connection. This function takes two arguments, the wire and the host address of the connecting process. See the section on host addresses below. When *on-connect* is `nil`, the request server allows all connections. When it is non-`nil`, the function returns two values, whether

to accept the connection and a function the system should call when the connection terminates. Either value may be **nil**, but when the first value is **nil**, the acceptance mechanism destroys the wire.

create-request-server returns an object that **destroy-request-server** uses to terminate a connection.

wire:destroy-request-server *server* [*Function*]

destroy-request-server takes the result of **create-request-server** and terminates that server. Any existing connections remain intact, but all additional connection attempts will fail.

wire:connect-to-remote-server *host port* **&optional** *on-death* [*Function*]

connect-to-remote-server attempts to connect to a remote server at the given *port* on *host* and returns a wire structure if it is successful. If *on-death* is non-**nil**, it is a function the system invokes when this connection terminates.

12.1.2. Remote Evaluations

After the server and client have connected, they each have a wire allowing function evaluation in the other process. This RPC mechanism has three flavors: for side-effect only, for a single value, and for multiple values.

Only a limited number of data types can be sent across wires as arguments for remote function calls and as return values: integers inclusively less than 32 bits in length, symbols, lists, and *remote-objects* (see section 12.1.3). The system sends symbols as two strings, the package name and the symbol name, and if the package doesn't exist remotely, the remote process signals an error. The system ignores other slots of symbols. Lists may be any tree of the above valid data types. To send other data types you must represent them in terms of these supported types. For example, you could use **prin1-to-string** locally, send the string, and use **read-from-string** remotely.

wire:remote *wire* {*call-specs*}* [*Macro*]

The **remote** macro arranges for the process at the other end of *wire* to invoke each of the functions in the *call-specs*. To make sure the system sends the remote evaluation requests over the wire, you must call **wire-force-output**.

Each of *call-specs* looks like a function call textually, but it has some odd constraints and semantics. The function position of the form must be the symbolic name of a function. **remote** evaluates each of the argument subforms for each of the *call-specs* locally in the current context, sending these values as the arguments for the functions.

Consider the following example:

```
(defun write-remote-string (str)
  (declare (simple-string str))
  (wire:remote wire
    (write-string str)))
```

The value of **str** in the local process is passed over the wire with a request to invoke **write-string** on the value. The system does not expect to remotely evaluate **str** for a value in the remote process.

wire:wire-force-output *wire* [*Function*]

wire-force-output flushes all internal buffers associated with *wire*, sending the remote requests. This is necessary after a call to **remote**.

wire:remote-value *wire call-spec* [*Macro*]

The **remote-value** macro is similar to the **remote** macro. **remote-value** only takes one *call-spec*, and it returns the value returned by the function call in the remote process. The value must be a valid type the system can send over a wire, and there is no need to call **wire-force-output** in conjunction with this interface.

If client unwinds past the call to **remote-value**, the server continues running, but the system ignores the value the server sends back.

If the server unwinds past the remotely requested call, instead of returning normally, **remote-value** returns two values, **nil** and **t**. Otherwise this returns the result of the remote evaluation and **nil**.

wire:remote-value-bind *wire ({variable}*) remote-form {local-forms}** [*Macro*]

remote-value-bind is similar to **multiple-value-bind** except the values bound come from *remote-form*'s evaluation in the remote process. The *local-forms* execute in an implicit **progn**.

If the client unwinds past the call to **remote-value-bind**, the server continues running, but the system ignores the values the server sends back.

If the server unwinds past the remotely requested call, instead of returning normally, the *local-forms* never execute, and **remote-value-bind** returns **nil**.

12.1.3. Remote Objects

The wire mechanism only directly supports a limited number of data types for transmission as arguments for remote function calls and as return values: integers inclusively less than 32 bits in length, symbols, lists. Sometimes it is useful to allow remote processes to refer to local data structures without allowing the remote process to operate on the data. We have *remote-objects* to support this without the need to represent the data structure in terms of the above data types, to send the representation to the remote process, to decode the representation, to later encode it again, and to send it back along the wire.

You can convert any Lisp object into a remote-object. When you send a remote-object along a wire, the system simply sends a unique token for it. In the remote process, the system looks up the token and returns a remote-object for the token. When the remote process needs to refer to the original Lisp object as an argument to a remote call back or as a return value, it uses the remote-object it has which the system converts to the unique token, sending that along the wire to the originating process. Upon receipt in the first process, the system converts the token back to the same (**eq**) remote-object.

wire:make-remote-object *object* [*Function*]

make-remote-object returns a remote-object that has *object* as its value. The remote-object can be passed across wires just like the directly supported wire data types.

wire:remote-object-p *object* [*Function*]

The function **remote-object-p** returns **t** if *object* is a remote object and **nil** otherwise.

wire:remote-object-local-p *remote* [*Function*]

The function **remote-object-local-p** returns **t** if *remote* refers to an object in the local process. This is can only occur if the local process created *remote* with **make-remote-object**.

wire:remote-object-eq *obj1 obj2* [*Function*]

The function **remote-object-eq** returns **t** if *obj1* and *obj2* refer to the same (**eq**) lisp object, regardless of which process created the remote-objects.

wire:remote-object-value *remote* [*Function*]

This function returns the original object used to create the given remote object. It is an error if some other process originally created the remote-object.

wire:forget-remote-translation *object* [*Function*]

This function removes the information and storage necessary to translate remote-objects back into *object*, so the next **gc** can reclaim the memory. You should use this when you no longer expect to receive references to *object*. If some remote process does send a reference to *object*, **remote-object-value** signals an error.

12.1.4. Host Addresses

The operating system maintains a database of all the valid host addresses. You can use this database to convert between host names and addresses and vice-versa.

ext:lookup-host-entry *host* [*Function*]

lookup-host-entry searches the database for the given *host* and returns a host-entry structure for it. If it fails to find *host* in the database, it returns **nil**. *Host* is either the address (as an integer) or the name (as a string) of the desired host.

ext:host-entry-name *host-entry* [*Function*]
ext:host-entry-aliases *host-entry* [*Function*]
ext:host-entry-addr-list *host-entry* [*Function*]
ext:host-entry-addr *host-entry* [*Function*]

host-entry-name, **host-entry-aliases**, and **host-entry-addr-list** each return the indicated slot from the host-entry structure. **host-entry-addr** returns the primary (first) address from the list returned by **host-entry-addr-list**.

12.2. The WIRE Package

The "**WIRE**" package provides for sending data along wires. The "**REMOTE**" package sits on top of this package.

All data sent with a given output routine must be read in the remote process with the complementary fetching routine. For example, if you send so a string with **wire-output-string**, the remote process must know to use **wire-get-string**. To avoid rigid data transfers and complicated code, the interface supports sending *tagged* data. With tagged data, the system sends a tag announcing the type of the next data, and the remote system takes care of fetching the appropriate type.

When using interfaces at the wire level instead of the RPC level, the remote process must read everything sent by these routines. If the remote process leaves any input on the wire, it will later mistake the data for an RPC request causing unknown lossage.

12.2.1. Untagged Data

When using these routines both ends of the wire know exactly what types are coming and going and in what order. This data is restricted to the following types:

- 8 bit unsigned bytes.
- 32 bit unsigned bytes.
- 32 bit integers.
- simple-strings less than 65535 in length.

wire:wire-output-byte *wire byte* [*Function*]
wire:wire-get-byte *wire* [*Function*]
wire:wire-output-number *wire number* [*Function*]
wire:wire-get-number *wire* **&optional** *signed* [*Function*]
wire:wire-output-string *wire string* [*Function*]
wire:wire-get-string *wire* [*Function*]

These functions either output or input an object of the specified data type. When you use any of these output routines to send data across the wire, you must use the corresponding input routine interpret the data.

12.2.2. Tagged Data

When using these routines, the system automatically transmits and interprets the tags for you, so both ends can figure out what kind of data transfers occur. Sending tagged data allows a greater variety of data types: integers inclusively less than 32 bits in length, symbols, lists, and *remote-objects* (see section 12.1.3). The system sends symbols as two strings, the package name and the symbol name, and if the package doesn't exist remotely, the remote process signals an error. The system ignores other slots of symbols. Lists may be any tree of the above valid data types. To send other data types you must represent them in terms of these supported types. For example, you could use **prin1-to-string** locally, send the string, and use **read-from-string** remotely.

wire:wire-output-object *wire object* **&optional** *cache-it* [*Function*]
wire:wire-get-object *wire* [*Function*]

The function **wire-output-object** sends *object* over *wire* preceded by a tag indicating its type.

If *cache-it* is non-**nil**, this function only sends *object* the first time it gets *object*. Each end of the wire associates a token with *object*, similar to remote-objects, allowing you to send the object more efficiently on successive transmissions. *Cache-it* defaults to **t** for symbols and **nil** for other types. Since the RPC level requires function names, a high-level protocol based on a set of function calls saves time in sending the functions' names repeatedly.

The function **wire-get-object** reads the results of **wire-output-object** and returns that object.

12.2.3. Making Your Own Wires

You can create wires manually in addition to the **"REMOTE"** package's interface creating them for you. To create a wire, you need a Unix *file descriptor*. If you are unfamiliar with Unix file descriptors, see secton 2 of the Unix manual pages.

wire:make-wire *descriptor* [*Function*]
The function **make-wire** creates a new wire when supplied with the file descriptor to use for the underlying I/O operations.

wire:wire-p *object* [*Function*]
This function returns **t** if *object* is indeed a wire, **nil** otherwise.

wire:wire-fd *wire* [*Function*]
This function returns the file descriptor used by the *wire*.

12.3. Out-Of-Band Data

The TCP/IP protocol allows users to send data asyncronusly, otherwise known as *out-of-band* data. When using this feature, the operating system interrupts the receiving process if this process has chosen to be notified about out-of-band data. The receiver can grab this input without affecting any information currently queued on the socket. Therefore, you can use this without interfering with any current activity due to other wire and remote interfaces.

Unfortunately, most implementations of TCP/IP are broken, so use of out-of-band data is limited for safety reasons. You can only reliably send one character at a time.

This routines in this section provide a mechanism for establishing handlers for out-of-band characters and for sending them out-of-band. These all take a Unix file descriptor instead of a wire, but you can fetch a wire's file descriptor with **wire-fd**.

wire:add-oob-handler *fd char handler* [*Function*]
The function **add-oob-handler** arranges for *handler* to be called whenever *char* shows up as out-of-band data on the file descriptor *fd*.

wire:remove-oob-handler *fd char* [*Function*]
This function removes the handler for the character *char* on the file descriptor *fd*.

wire:remove-all-oob-handlers *fd* [*Function*]
This function removes all handlers for the file descriptor *fd*.

wire:send-character-out-of-band *fd char* [*Function*]
This function Sends the character *char* down the file descriptor *fd* out-of-band.

Index

Index